OPERATING SYSTEM LEVEL RESOURCE MANAGEMENT FOR REAL-TIME SYSTEMS

BY

HEECHUL YUN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair
Professor Marco Caccamo
Professor Tarek Abdelzaher
Professor Rodolfo Pellizzoni, University of Waterloo

ABSTRACT

The main goal of a real-time operating system (RTOS) is to provide foundations for guaranteeing deterministic or probabilistic timing requirements of real-time systems. In that regard, scheduling the CPU time has been the core aspect of any RTOS, as it directly contributes to satisfying timing requirements of real-time systems.

In recent years, however, the emergence of multicore processor architecture and the growing number of battery powered devices make other aspects—such as contention in concurrently shared hardware resources and power consumption—also critical in providing required real-time performance. Management of concurrently shared hardware resources—such as shared caches, DRAM controllers, DRAM modules, hardware prefetchers—is important because contention in the shared resources can significantly degrade applications' execution times. Power (energy) management is also important because power-saving techniques typically have significant performance implications.

This dissertation describes new OS level mechanisms and policies that control the shared hardware resources in commodity processors in ways that improve performance isolation and reduce energy consumption of real-time systems. These techniques enhance a RTOS's resource management capability so that it can be flexibly tailored to meet challenging real-time requirements without significantly compromising overall performance or wasting energy.

Contributions of this dissertation include the following: (1) The design, prototype implementation, and performance evaluation of an efficient fine-grained memory bandwidth reservation system called MemGuard that improves performance isolation of multicore based real-time systems. (2) The design, model validation, and performance evaluation of the MultiDVFS scheme that jointly optimizes CPU and memory frequencies/voltages for real-time systems.

To my family, for their love and support.

ACKNOWLEDGMENTS

The work presented in this dissertation is the culmination of years of my PhD research. First of all, I would like to thank my advisor Prof. Lui Sha. He has been patient, accessible, and willing to help me on various issues. I am also grateful to other members of my PhD committee—Prof. Marco Caccamo, Prof. Tarek Abdelzaher, and Prof. Rodolfo Pellizzoni—for their guidance and feedback in developing my research. I am also very thankful to my collaborators who contributed to the presented work in many ways. I thank Cheolgi Kim; he has been a very good mentor whom I can discuss any issues. I thank Gang Yao; he has been a great research partner as well as a jogging partner for me—I really enjoyed the regular once-a-week jogging with him. Also, I also thank Po-Liang Wu and Anshu arya; working with them on a paper was one of the most memorable and rewarding moments in my PhD.

Above all, I thank my family, for their love and support. My two precious sons, Yejun and Sejun, made me laugh even during the difficult times. I am indebted to my wife Jeongsuk Kim for her dedication and sacrifices; without her support, it would have been very difficult for me to complete my PhD program.

Portions of this dissertation are reprinted, with permission, from the following publications:

- "MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms", H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Real-Time and Embedded Technology and Applications Symposium (RTAS), ©2013 IEEE

- "Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality", H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, L. Sha, Euromicro Conference on Real-Time Systems (ECRTS), ©2012 IEEE

- "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks", H. Yun, P. Wu, A. Arya, T. Abdelzaher, C. Kim, L. Sha, Real-Time Systems, ©2011 Springer

- "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks", H. Yun, P. Wu, A. Arya, T. Abdelzaher, C. Kim, L. Sha, Euromicro Conference on Real-Time Systems (ECRTS), ©2010 IEEE

The material presented in this thesis is based upon work supported in part by ONR N00014-12-1-0046, Lockheed Martin 2009-00524, Rockwell Collins RPS#6 45038 and RPS#1 2007-05974, and NSF CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the funding agencies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The main goal of a real-time operating system (RTOS) is to provide foundations for guaranteeing deterministic or probabilistic timing requirements of real-time systems. One of the most important factors affecting the temporal aspect of a real-time system is how the CPU time is multiplexed—i.e., CPU scheduling—among multiple applications in the system. Therefore, CPU scheduling has been a core functionality of any existing RTOSs.

In recent years, however, underlying processor architecture of computing systems is experiencing a paradigm shift to more power efficient, more integrated, and multicore based architecture. Because many modern embedded systems, such as Unmanned Aerial Vehicle (UAV) systems, need both high performance and low-power consumption, such transition can be a desirable thing for the real-time embedded system community. The transition, however, raises enormous challenges for RTOSs in providing necessary real-time guarantees.

1.1 Resource Management Challenges

In modern multicore architecture, many hardware resources—e.g., shared cache, hardware prefetchers, and memory controller—are shared among the cores. While sharing such resources generally improves overall efficiency, contention in the shared resources can significantly increases task execution times, often by multiple factors as reported by many researchers [2, 3, 4, 5, 6]. Such high variation in execution times is a serious problem for critical real-time embedded systems as timing properties of applications can be altered by other unrelated co-running applications on other cores. The impact of resource contention can be even more severe in the future, as the number of cores that share the resources increases.

Furthermore, the ubiquity of battery powered devices (such as mobile phones, tablets, and smart sensors) and concerns over energy bill in sever systems make CPU vendors to invent various power saving techniques such as dynamic voltage frequency scaling (DVFS). These power saving techniques, however, generally have significant performance implication, further complicating RTOSs in providing adequate real-time performance.

1.2 Contributions

In this thesis, I propose new OS level mechanisms and policies that control shared hardware resources in commodity multicore processors in ways that improve performance isolation and reduce energy consumption of real-time systems.

First, I propose an OS level memory bandwidth management system that provides memory bandwidth reservation among concurrently accessing cores in multicore systems. The solution has been implemented in a real OS (Linux) kernel and has demonstrated that significantly improve performance isolation in real multicore platforms. I also provide an analytic framework to analyze worst-case schedulability of the task system under a memory bandwidth regulated system.

Second, I propose a dynamic voltage and frequency scaling (DVFS) technique that jointly adjust both CPU and memory frequencies to reduce energy consumption while meeting the deadlines. We present both simulation and experimental results on a real hardware platform that demonstrate its effectiveness in conserving energy while satisfying real-time requirements.

I believe the proposed solutions are important foundations for current and future RTOSs as they provide flexible and powerful tools to deliver adequate real-time performance on commodity hardware platforms.

1.3 Organization

Chapter 2 reviews necessary background and previous work.

Chapter 3 presents a memory bandwidth management system providing memory performance isolation on multicore platforms.

Chapter 4 presents a response time analysis framework for memory bandwidth regulated multicore platforms.

Chapter 5 presents a frequency assignment scheme that manages multiple DVFS capable components for real-time systems.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, we describe some necessary background about modern processor architecture and discuss challenges for real-time operating systems (RTOSs). We also review past work on managing shared hardware resources and power in the context of real-time systems.

2.1 Real-Time Operating Systems

Real-time systems can be defined as the systems that have deterministic or probabilistic timing requirements. To satisfy these requirements, the CPU scheduling algorithm and its analysis technique has been one of the most important factors in the design of a real-time system. The CPU scheduler, therefore, has been a key component in any RTOS [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20].

The most common CPU scheduling algorithm for a RTOS is the fixed priority (FP) based preemptive scheduling algorithm. It is easy to implement in an OS and the rate-monotonic (RM) priority assignment policy and its analysis methods has been successfully applied to many industries, such as avionics [21]; therefore, most commercial RTOSs are primarily based on FP schedulers [12, 13, 14, 15] as well as many academic RTOSs [7, 8, 10, 11].

The earliest-deadline-first (EDF) algorithm is another well-established CPU scheduling algorithm, especially in academia. Unlike FP, it dynamically change priority of a task based on its deadline so that the most urgent task will be scheduled which offers the optimal CPU utilization bound. Despite its popularity in academia, its adoption in commercial RTOSs has been very limited. One reason is that it is more complex to implement in an OS. Another, perhaps more important, reason is that it performs poorly in an overload situation while FP shows more robust behavior in such a situation.

The EDF scheduler has been implemented in several academic RTOSs [16] and more recently in Linux [17, 18, 19, 20] and the Xen hypervisor [22].

Some RTOSs additionally employ partition scheduling algorithms [12, 13, 14, 15]. A partition scheduling algorithm divides time into a set of partitions for stronger temporal protection between partitions. Inside each partition, however, tasks are scheduled using a traditional priority based scheduling algorithm. In avionics systems, partition scheduling is required [23], and there are many other use-cases that such scheduler can be beneficial [24].

For a chosen RTOS (and its scheduling algorithm), it is possible to perform a timing analysis if each real-time task's worst-case execution time (WCET) is known for the interested real-time system. Therefore, it is important to correctly identify the proper WCETs: Too optimistic WCETs would risk violating real-time requirements; too pessimistic WCETs would under-utilize computing resources. For reasons such as out-of-order execution and branch prediction, it has been a hard problem already in unicore based processors [25], but it is even more difficult problem in modern multicore based processors, in which even more sharing and concurrent interferences must be accounted. We now discuss the challenges that affect task execution times on modern embedded processors in the subsequent subsection.

2.2 Modern Multiprocessor System-On-Chip

The defining characteristics of modern processor architecture can be summed as multicore and power consumption. In the middle of the first decade of the 21st century, increasing clock speed to improve performance has reached its limit due to thermal and power constraints [26]. Since then, chip manufacturers began to add more cores in a chip to gain more performance. Increasing popularity of mobile devices and the concerns about energy efficiency in datacenter server systems further motivate the chip manufacturers to develop more energy efficient architecture. As a result, modern processor architecture integrates multiple processing cores, multiple levels of caches and memory controllers into a single chip (called multiprocessor system-on-a-chip or MPSoC) to provide more performance and consume less power.

Figure 2.1 shows the block diagram of a MPSoC, which is used in mobile devices such as tablets and smartphone systems. It integrates four ARM9

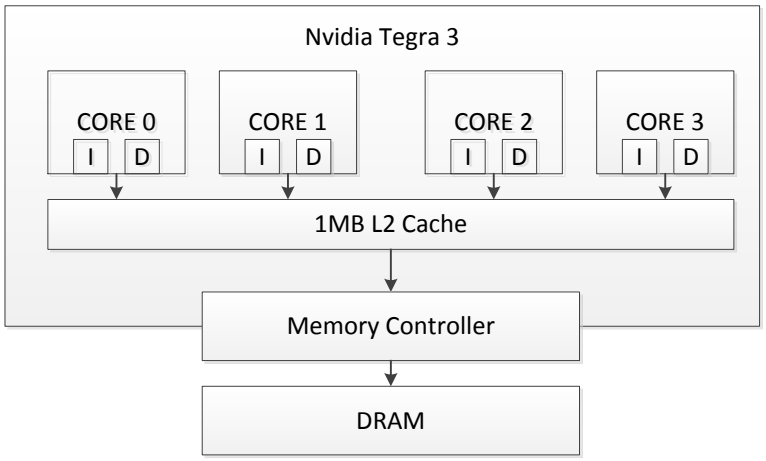


Figure 2.1: A typical multicore processor (Nvidia Tegra3 processor).

cpu cores, a 2MB shared L2 cache, a memory controller, GPU cores, and many other special hardware blocks, all in a single processor die offering great computational power for embedded systems, while only consumes less than 2 watt of power [26].

Improved throughput and high energy efficiency of modern MPSoCs can benefit many high-end real-time embedded systems. For example, a Unmanned Aerial Vehicle (UAV) that demand high computing performance can minimize size, weight, and power consumption—known as SWaP properties—by using such an efficient MPSoC processor [27, 28]. There are, however, significant challenges that should be addressed by RTOSs for applying them to real-time systems.

The first challenge is to understand and manage performance/execution time impact of shared micro-architectural hardware resources in modern SoCs. This is crucial because guaranteeing timing is a key requirement for real-time systems. Many researchers have shown that contentions in shared resources can dramatically degraded applications' performance on multicore processors [2, 29, 5, 30, 6, 4, 3]. First, it is well known that the space competition in shared caches can cause significant performance degradation because a task's cache entries can be evicted by other unrelated tasks on different cores. For example, Kim et al. showed it can cause up to 9.5x increase in last level cache (LLC) and 2.7X execution time slowdown in a dual-core CMP[2]. Mancuso et al. also showed up to 2.5X WCET increase

due to cache contention in a dual core ARM processor [30]. Shared DRAM controllers (and connected DRAM devices) are another important sources of contention. In fact, Zhuravlev et al. showed the contention in accessing data in DRAM influenced more than the contention in shared cache for many SPEC benchmarks [4]. Other studies that investigated the impact of DRAM controller sharing (without using shared caches) [6, 1, 31, 32] also clearly showed its significant performance impact. For example, Mutlu et al. showed up to 5.2X slowdown due to the contention in the DRAM controller in a dual-core platform [1]. Such high performance impact—multiple factors of slowdown—is generally unacceptable for real-time systems where execution times of critical real-time tasks must be guaranteed. For example, to analyze safety aspects, Avionics software platforms are required to provide isolated execution environments, called partitions, that any unintentional interaction between different partitions must be avoided [23, 27]. It is, however, difficult to meet when WCET can increase by several factors by activities on other cores which run different partitions.

The second challenge is how a RTOS can minimize power/energy consumption of a real-time system without violating its real-time requirements. While processor manufacturers are busy reducing power consumption of their processors by means of various hardware techniques (e.g., manufacturing process improvement and automatic power-gating), they also expose several mechanisms—namely, dynamic voltage/frequency scaling and multiple sleep states, known as DVFS and DPM respectively—to RTOS designers so that they can be tuned more effectively leveraging OS and task level information that is not available to hardware. While these mechanisms can save power and energy consumption, they also have significant performance implications.

The fact that shared hardware resources, such as DRAM modules, consume significant power, which is comparable to CPU as shown in [33], and often provide power saving mechanisms, similar to CPU cores, further complicates the problem.

Therefore a great care must be taken in utilizing these power saving mechanisms to reduce energy consumption of real-time systems.

2.3 Shared Resource Management

CPU scheduling: In a unicore processor, only one task runs at a time and it can exclusively use the entire hardware resources in the processor until it is preempted by another task. Therefore, the amount of time a task is scheduled on the CPU directly determines the amount of computation it perform, excluding overhead involved in task switching (including cache and pipeline effects). Based on this premise, schedulability of periodic real-time tasks using either a static priority or a dynamic priority based scheduler can be analyzed, assuming each task's period and WCET are known [34, 35]. Aperiodic tasks can be accommodated into the analysis of periodic task systems by running them through real-time servers such as Sporadic Server [36] and Constant Bandwidth Server (CBS) [37] for RM and EDF respectively.

Scheduling theories and RTOS implementations have long been extended to support multi processors [38, 39, 18, 14, 15, 12]. However, these results generally do not consider contention in shared hardware resources, which can increase tasks' execution times by several factors.

Cache management: In the real-time community, managing shared cache in multicore platforms has been a major research focus in recent days to address the problem of execution time impact due to cache space contention in order to predictability [30, 40]. For more general purpose computing systems, cache space contention has been extensively studied in order to improve fairness and throughput [41, 2, 29, 42, 43]. The majority of the work focuses on space competetion by partitioning the cache space appropriately, with an exception of [41] which focuses on bandwidth compeition in shared caches. In partitioning cache space, a popular technique is *Page Coloring*. Page coloring is an OS level technique using MMU hardware block; it divides the physical address space into a small number of colors (the number of available colors is determined by the specific cache organization.) and the pages having different colors can not evict each other from the cache. By carefully assigning the colors in allocating pages, the OS can avoid cache eviction problem. This technique can be applied to any hardware platform having MMU hardware. Please refer [30, 42] for more detailed explanation about page coloring. Other techniques are based on hardware supported cache-way/line locking mechanisms. Compared to page coloring, these hardware supported techniques

can provide more fine-grain control but they are architecture specific. For example, Intel processors do not support any of these locking mechanisms while many embedded processors from ARM and Freescale support these features [44, 45, 46].

Recent work by Mancuso et al. proposed a framework that leverages cache coloring and locking to flexibly manage cache space for hard real-time applications on multicore platforms [30]. Ward et al. also utilized coloring and locking disciplines and proposed a cache scheduling method for predictable real-time system [40].

For more general purpose systems, Zhang et al. proposed a page coloring based technique to improve fairness in multi programmed multicore environment [42]. Ding et al. also proposed a page coloring based technique to minimize destructive cache pollution due to unproductive buffer cache usage [43]. Another large body of work proposed various hardware level modifications of shared cache to improve fairness and throughput [29, 41, 2].

DRAM management: For improving performance predictability in main memory (DRAM), bandwidth and access latency are generally considered more important than space competition because the DRAM size is typically bigger than the size of applications of a real-time system. Most work, therefore, focuses on bandwidth and latency aspects of DRAM. DRAM bandwidth is different from CPU bandwidth in the sense that achievable bandwidth depends on how multiple memory resources, called banks, are utilized. Likewise, DRAM access latency also varies depending on the states of DRAM chips and scheduling at DRAM controllers (More details concerning inner working of DRAM can be found in Chapter 3.) It is, therefore, difficult to provide performance guarantees in accessing DRAM. To solve this problem, several predictable DRAM controllers were proposed for real-time systems [47, 48, 49, 50].

Akesson et al. proposed Predator DRAM controller that provides bandwidth guarantees for multiple requesting hardware components [47]. It uses a close-row policy and, it accesses all DRAM banks simultaneously, for each memory request, in order to eliminate DRAM state dependent access latency variation without losing too much bandwidth. In order to provide latency and bandwidth guarantees, it uses an arbitration scheme, called CCSP [51], which combines a version of latency-rate (LR) server [52] and a static priority based scheduler. Paolieri et al. also proposed a memory controller design,

called AMC, for hard real-time systems [48]. It also uses a close-row policy and accesses all banks at a time to eliminate DRAM state dependent latency variation. Its arbitration scheme is, however, different in that it uses a round-robin arbitration scheme. They argue that the round-robin arbitration is better suited for typical hard real-time tasks, while the CCSP arbiter may be better suited for stream oriented multimedia workloads.

Both AMC and Predator use a close-row policy and access all banks at a time to achieve both predictability and good bandwidth utilization. The downside of this approach is, however, the access granularity can be much bigger than the cache-line size of the CPU, hence potentially wasting bandwidth. To solve this problem, Zheng et al. [50] proposed a new DRAM controller that uses a open-row policy and per-core private bank mapping. Because row-hit memory accesses in the open-row policy is much faster than row-miss accesses, the average WCET characteristic is improved especially for those have high row-hit ratio. The limitation of this approach is, however, cores cannot share memory due to private banking, preventing its adoption in commodity multi-core hardware platforms. Reineke et al. also adopted a private banking based approach in their PRET DRAM controller [53]. This work is intended to be used in PRET architecture [54, 55] that is designed from the ground up to support time predictability at the level of hardware architecture.

In more general purpose computing systems, DRAM controllers are studied in order to improve fairness and throughput. Nesbit et al. applied the network fair queuing theory in designing DRAM controller [32]. Rafique et al. also applied a different version of fair queuing algorithm, namely start-time fair queuing, and they also proposed a feedback-based adaptive policy to maintain desired memory access latency [56]. Multlu et al. proposed DRAM controller designs that employ several different heuristic algorithms for improving fairness [1, 31].

While these DRAM controller designs provide solutions to improve predictability, isolation, and fairness at the hardware level, the focus of this thesis is software based solutions that can be applied in commodity hardware platforms.

Resource sharing aware scheduling: In handling the challenges arise from the contention in shared resources, another line of research is resource sharing aware CPU scheduling. The basic idea is to take the effect of re-

source sharing into account in making scheduling decisions in order to improve predictability (for real-time systems), and/or fairness and throughput (for general purpose systems).

Modern MPSoCs equip hardware performance counters that can be configured to measure various micro-architectural statistics such as cache-misses and retied instructions. By utilizing these counters, schedulers can gather many useful data to make better scheduling decisions. There are generally two different scheduling goals: *space control* and *rate control*.

First, controlling space is about deciding which task to run which core. For example, one may want to schedule a "light" task (which uses small cache space) and a "heavy" task (which uses large cache space) on a dual core that share a LLC, instead of scheduling two heavy tasks. This kind of scheduling techniques were first emerged for efficient scheduling on Simultaneous Multithreading (SMT) processors [57, 58], and more recently for Chip Multiprocessors (CMP) [1] [59, 3, 4, 60]. An in-depth survey of various proposals can be found in [61]. Generally, their main goal is to find a mapping between tasks and cores in the system that minimizes contention in the given shared resources, hence maximizing overall throughput. In the real-time system community, Calandrino et al. adopted a space control approach [62, 63]. In this work, the goal is to find a subset of tasks that can be efficiently co-scheduled in multiple cores—i.e., not causing cache eviction of each other—while still meeting real-time constraints of all tasks. They proposed several heuristic algorithms [62] and later implemented their best heuristic algorithm in Linux [63], demonstrating practical effectiveness of the approach.

Another approach for a RTOS is controlling rate of each task in the system. By controlling the rate of a task, we mean controlling the task's execution speed as a mean of indirectly controlling the task's access rate of shared resources. It can be done in several ways: One way is leveraging hardware features such as DVFS and duty-cycle modulation [64]; another way is developing software level methods, likely at OS kernel level, to control speed of execution [65, 6]. Either way, the basic idea is if a task executes slowly, its use of shared resources also reduces, consequently cause less contention to competing tasks that share the resources. Bellosa [65, 66] was the first

[1] we use the terms "CMP" and "MPSoC" interchangeably

to propose this approach. He modified a TLB miss handler to slowdown the execution speed (by adding idle loops inside the handler) in order to provide more memory bandwidth for multimedia applications. More recently, Herdrich et al. proposed a rate control based approach which uses both DVFS and CPU duty cycle modulation mechanisms to slowdown the low priority tasks' progresses [64]. Ebrahimi et al. proposed a cache controller level throttling mechanism for improving fairness [67] among competing cores.

2.4 Power Management

Dynamic Voltage/Frequency Scaling: There is a significant amount of previous work focusing on static DVFS schemes for real-time tasks under RM or EDF scheduling [68, 69, 70, 71]. Aydin and Melhem [71] formulated an energy minimization problem and proposed an algorithm to find the optimal static frequency assuming that individual tasks that have different power characteristics. Jejurikar and Gupta [70] proposed an energy model in which the deadline was not equal to the period, and presented methods for finding weak-optimal slowdown factors for scheduling periodic tasks.

On-line slack time reclamation has been another realm of DVFS research. Pillai and Shin [72] proposed dynamic reclamation technique (cycle-conserving and look-ahead) which exploits unused slack time dynamically to save energy. Gruian et al. also proposed a dynamic algorithm that utilizes slack time to lower energy usage and employs both on-line and off-line scheduling policies [69]. Mejia-Alvarez et al. [68] proposed an on-line scheduling algorithm with discrete frequency steps for DVFS. Zhong et al. proposed an analytical model for scheduling without prior task information [73].

Although CPU cores consumes a significant amount of energy, other components such as the main memory and system bus often consume energy on a similar order of magnitude [33]. Without considering such components, energy savings may not be maximized. Several researchers considered main-memory accesses [74, 75, 76, 77, 78] because memory access time does not scale with the CPU frequency. Bini et al. [74, 75] proposed an energy model that accounts for memory operations in the task execution time when only CPU frequency is adjustable. Aydin et al. [77] proposed a similar energy model that also considered CPU frequency independent power components.

Cho et al. [79] proposed an energy model that considered both CPU and memory frequencies but did not consider real-time tasks. Snowdon et al. also proposed an execution time and energy model [80, 81] that considered multiple adjustable frequencies and task characteristics but they also did not consider real-time tasks.

Dynamic Power Management: Dynamic Power Management (DPM) is another widely adopted mechanism to reduce power consumption which turns off system components during idle periods [82]. Several works proposed to combine DVFS with DPM to achieve higher energy efficiency. Simunic et al. [83] proposed an algorithm to merge the DPM and the DVFS approaches based on the stochastic model. Liu et al. [84] proposed an algorithm combining both DVFS and DPM for streaming applications on embedded multiprocessors. Cheng et al. [85] presented a system-wide energy-aware EDF (SYS-EDF) algorithm, which integrates DPM for I/O devices and DVFS for the processor. Rong et al. [86] proposed a three-phase solution framework to reduce the system-wide power consumption with the consideration of task dependencies for periodic hard real-time tasks.

Some previous work addresses power-saving with component standby modes in addition to a DVFS-capable CPU. Zhuo et al. [76] proposed a system-wide energy model that considered standby mode components and showed that as the number of components grew, the effectiveness of DVFS schemes decreased due to increased standby power. Zhong et al. [73] also presented an energy model accounting for standby modes for both periodic and sporadic tasks. Devadas et al. proposed Device Forbidden Regions approach to explicitly enforce long sleep intervals for different system devices [87]. Exploiting a look-ahead technique, the algorithm postpones the forbidden region when it is possible to achieve more energy savings.

CHAPTER 3

MEMORY BANDWIDTH RESERVATION

As introduced in the previous chapter, contention in shared resources is a challenging problem for the design of a real-time OS (RTOS), which is responsible delivering QoS to each task. In this chapter, we focus on shared memory bandwidth[1]. We assess the impact of memory bandwidth sharing and propose an OS level mechanism in order to provide efficient performance isolation guarantees.

Computing systems are increasingly moving toward multicore platforms and their memory subsystem represents a crucial shared resource. As applications become more memory intensive and more cores share the same memory system, the performance of main memory becomes more critical for overall system performance.

In a multicore system, the processing time of a memory request is highly variable as it depends on the location of the access and the state of DRAM chips and the DRAM controller. Requests from different cores can change the state of DRAM chips as well as queuing delay in the DRAM controllers. Furthermore, DRAM controllers commonly employ scheduling algorithms to re-order requests in order to maximize overall DRAM throughput [32]. All these factors make it difficult to provide predictable performance for memory intensive real-time applications.

Therefore, there is an increasing need for solutions that provide Quality of Service (QoS) on accessing main memory. In the real-time community, there has been a series of work proposing predictable DRAM controllers [47, 48, 49, 50]. More generally, resource reservation and reclaiming techniques [88, 37] have been widely studied by the real-time community to solve the problem of assigning different fractions of a shared resource in a guaranteed manner to contending applications. These techniques have been successfully applied to

[1]Unless noted otherwise, we use terms "memory" and "DRAM" interchangably throughout this thesis

CPU management [89, 90, 91, 92] and more recently to GPU management [93, 94].

Unfortunately, existing solutions cannot be easily used for managing memory bandwidth. First, predictable DRAM controllers are not readily available in Commercial Off-The-Shelf (COTS) components, as of writing. As increasing number of real-time systems are built with COTS components, we need solutions that can be applied to COTS components. Second, CPU resource reservation solutions [90, 95, 92] cannot be directly applied to the memory system because 1) the achievable memory service rate is highly dynamic, as opposed to the constant service rate in CPU scheduling, and 2) RTOSs generally do not have exact knowledge of when a task access memory.

In this chapter, we present an OS level memory bandwidth reservation system, which we call *MemGuard*. Unlike CPU bandwidth reservation, under MemGuard the available memory bandwidth can be described as having two components: *guaranteed* and *best effort*. The guaranteed bandwidth represents the minimum service rate the DRAM system can provide, while the additionally available bandwidth is best effort and cannot be guaranteed by the system. Memory bandwidth reservation is based on the guaranteed part in order to achieve temporal isolation. Furthermore, the guarantee is fine-grained in the sense that the reserved bandwidth is guaranteed for each scheduler tick interval. As argued in [92], fine-grain bandwidth guaranteed is important for time sensitive applications (e.g., multimedia applications or stock trading systems).

To efficiently utilize the guaranteed memory bandwidth, a reclaiming mechanism is proposed leveraging each core's usage prediction. The system throughput is further improved by exploiting the best effort bandwidth after the guaranteed bandwidth of each core is satisfied. Since our reclaiming algorithm is prediction based, misprediction can lead to a situation where guaranteed bandwidth is not delivered to the core. Therefore, predictive reclaim is intended to support mainly soft real-time systems. For hard real-time tasks, MemGuard management framework should be used with the reclaiming feature disabled. We evaluate the performance of MemGuard under different configurations and we present detailed results in the evaluation section.

In summary, the contributions of this work are: (1) decomposing overall memory bandwidth into a guaranteed and a best effort component. Then, we experimentally identify the boundary so we can apply the proposed reserva-

tion technique; (2) designing and implementing (in Linux kernel) an efficient memory bandwidth reservation system, named MemGuard; (3) evaluating MemGuard with an extensive set of realistic SPEC2006 benchmarks [96] and showing its effectiveness on a real multicore hardware platform.

This chapter is organized as follows: Section 3.1 describes the challenge of predictability in modern multicore systems. Section 3.2, 3.3, and 3.4 describes the details of the proposed MemGuard approach. Section 3.5 describes the evaluation platform and the software implementation. Section 3.6 presents the evaluation results. Section 3.7 discusses related work. We conclude in Section 3.8.

3.1 Problems of Shared Memory in Multicore Systems

Many modern embedded systems process vast amount of data that are collected from various type of sensing devices such as surveillance cameras. Therefore, many real-time applications are increasingly becoming more memory bandwidth intensive. This is especially true in multicore systems, where additional cores increase the pressure on the shared memory hierarchy. Therefore, task execution time is increasingly more dependent on the way that memory resources are allocated among cores. To provide performance guarantees, real-time system have long adopted resource reservation mechanisms. Our work is also based on this approach but there are difficulties in applying reservation solution in handling memory bandwidth. To illustrate the problems, we performed two set of experiments on a real multicore hardware (described in Section 3.5.1).

In the first experiment, we measured Instructions-Per-Cycle (IPC) for each SPEC2006 benchmark (foreground task) first running on a core in isolation, and then together with a memory intensive benchmark (470.lbm) running on a different core (background task). Figure 3.1 shows IPC slowdown ratio (run-alone IPC/co-scheduled IPC) of both foreground and background tasks; foreground tasks are arranged from the most memory intensive to the least memory intensive on the X-axis. As clearly showed in the figure, both the foreground and the background task suffer slowdown since they are interfering each other in the memory accesses, which is expected. Interestingly, however, most of the times the foreground task slowdown more than

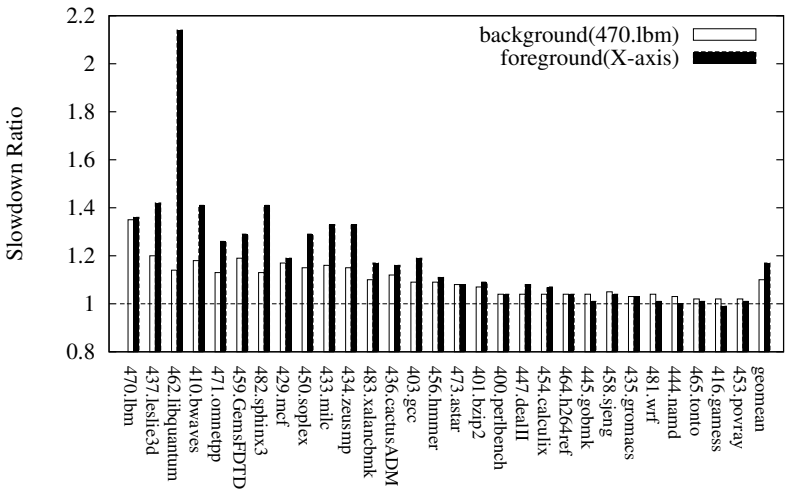
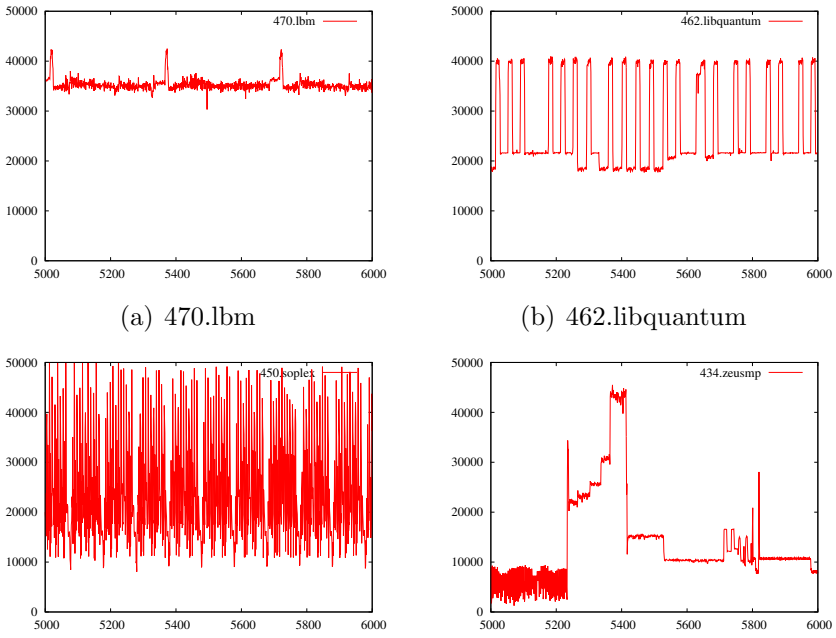


Figure 3.1: IPC slowdown of foreground (X-axis) and background task (470.lbm) on a dual-core configuration

the background task, and the difference of slowdown factors between that two tasks could be as large as factor of two (2.2x against 1.2x). Furthermore, note that the slowdown factor is not necessarily proportional to how memory intensive the foreground task is. Such effects are typical in COTS systems due to the characteristics of modern DRAM controllers [32]: (1) each DRAM chip is composed of multiple resources, called banks, which can be accessed in parallel. The precise degree of parallelism can be extremely difficult to predict, since it depends, among others, on the memory access patterns of the two tasks, the allocation of physical addresses in main memory, and the addressing scheme used by the DRAM controller. (2) Each DRAM bank itself comprises multiple rows; only one row can be accessed at a time, and switching row is costly. Therefore, sequential accesses to the same row are much more efficient that random accesses to different rows within a bank. DRAM controllers commonly implement scheduling algorithms that re-order requests depending on the DRAM state and the backlogged requests inside the DRAM controller, in order to maximize throughput [32]. 470.lbm tends to suffer less slowdown than foreground tasks because it is memory intensive and it floods the request queue in the DRAM controller with sequential access requests. These results indicate that memory bandwidth is very different compared to CPU bandwidth, in the sense that maximum achievable



(a) 470.lbm (b) 462.libquantum

(c) 450.soplex (d) 434.zeusmp

Figure 3.2: Memory access pattern of four representative SPEC2006 benchmarks.

bandwidth is not fixed but highly variable depending on access location of each memory request and on the state of DRAM subsystem.

Furthermore, the memory access patterns of tasks can be highly unpredictable and significantly change over time. Figure 3.2 shows the memory access patterns of four benchmarks from SPEC2006. Each benchmark runs alone and we collected memory bandwidth usage using hardware Performance Measuring Counters (PMC) between time 5 to 6 seconds, sampled over every 1ms time interval. 470.lbm shows highly uniform access pattern throughout the whole time. On the other hand, 462.libquantum and 450.soplex show highly variable access pattern, while 434.zeusmp show mixed behavior over time.

When resource usage changes significantly over time, a static reservation approach results in poor resource utilization and poor performance. If the resource is reserved with the maximum request, it would lead to significant waste as the task usually does not consume that amount; while if it is with the average value, the task would suffer slowdown whenever it tries to access more than that average value during one sampling period. The problem is only compounded when the available resource amount also changes over

time, as it is the case for memory bandwidth. One ideal solution is to dynamically adjust the resource provision based on its actual usage: when the task is highly demanding on the resource, it can try to reclaim some possible spare resource from other entities; on the other hand, when it consumes less than the reserved amount, it can share the extra resource with other entities in case they need. Furthermore, if the amount of available resource is higher than expected, we can allocate the remaining resource units among demanding tasks. There have been two types of dynamic adaptation schemes: feedback-control based adaptive reservation approaches [97] and resource reclaiming based approaches [91, 89]. In our work, we choose a reclaiming based approach for simplicity.

The details of our bandwidth management system, MemGuard, are provided in the next section. In summary, based on the discussed experiments, the system will need to: (1) reserve memory bandwidth resource to one specific core (a.k.a. resource reservation) to provide predictable and guaranteed worst-case behavior; and (2) provide some dynamic resource adjustment on the resource provision (a.k.a. resource reclaiming) to efficiently exploit varying system resources and improve task responsiveness.

3.2 MemGuard Overview

The goal of MemGuard is to provide memory performance isolation while still maximizing memory bandwidth utilization. By memory performance isolation, we mean that the average memory access latency of a task is no larger than when running on a dedicated memory system which processes memory requests at a certain service rate (e.g., 1GB/s). A multicore system can then be considered as a set of unicore systems, each of which has a dedicated, albeit slower, memory subsystem. This notion of isolation is commonly achieved through resource reservation approaches in real-time literature [37] mostly in the context of CPU bandwidth reservation.

We focus on systems that the last level cache is private or partitioned on a per-core basis, in order to focus on DRAM bandwidth instead of cache space contention effects. We assume system operators or an external user-level daemon will configure MemGuard either statically or dynamically via the provided kernel filesystem interface.

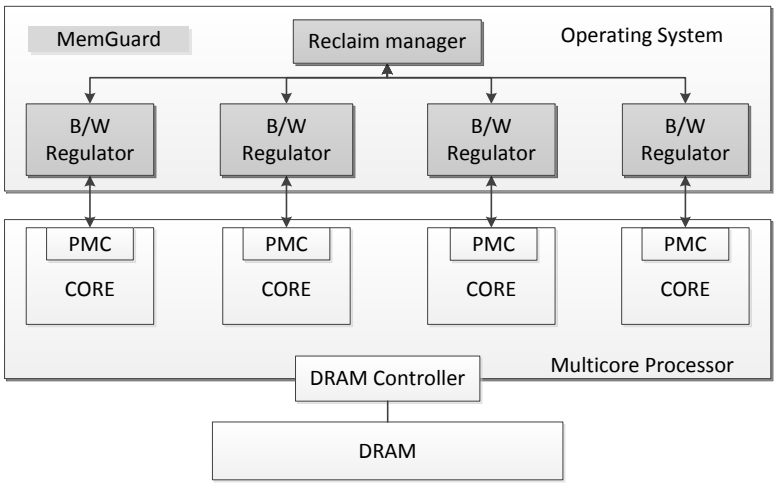


Figure 3.3: MemGuard system architecture.

Figure 3.3 shows the overall system architecture of MemGuard and its two main components: the per-core regulator and the reclaim manager. The per-core regulator is responsible for monitoring and enforcing its corresponding core memory bandwidth usage. It reads the hardware PMC to account the memory access usage. When the memory usage reaches a pre-defined threshold, it generates an overflow interrupt so that the specified memory bandwidth usage is maintained. Each regulator has a history based memory usage predictor. Based on the predicted usage, the regulator can donate its budget so that cores can start reclaiming once they used up their given budget. The reclaim manager maintains a global shared reservation for receiving and re-distributing the budget for all regulators in the system.

Using the regulators, MemGuard allows each core to reserve a fraction of memory bandwidth, similar to CPU bandwidth reservation. Unlike CPU bandwidth, however, available memory bandwidth varies depending on memory access patterns. In order to achieve performance isolation, MemGuard restricts the total reservable bandwidth to the worst case DRAM bandwidth—we call it *guaranteed bandwidth*, r_{min}; see Section 3.3—similar to the requirement of CPU bandwidth reservation in that the total sum of CPU bandwidth reservation must be equal or less than 100% of CPU bandwidth. Our observation is that it greatly reduces contention in the DRAM controller, hence significantly improve performance isolation. While reservation provides performance isolation, reclaiming improves overall resource utilization

by re-distributing under-utilized memory bandwidth to demanding cores.

While the reservation and reclaiming achieve performance isolation and efficient utilization of the guaranteed bandwidth, the guaranteed bandwidth is much smaller than the peak bandwidth of the given DRAM module. We call the difference (i.e., peak - guaranteed bandwidth) as *best-effort bandwidth*. MemGuard tries to utilize the best-effort bandwidth whenever possible.

Figure 3.4 shows the high level implementation of MemGuard. Using the figure as a guide, we now describe the memory bandwidth management mechanisms of MemGuard in the following sections: Section 3.3 describes how MemGuard manage the guaranteed bandwidth in more detail; Section 3.4 describes two different methods to utilize the best-effort bandwidth.

3.3 Guaranteed Bandwidth Management

In this section, we first give necessary background information about DRAM based memory subsystems and define the guaranteed bandwidth which is the basis for our reservation and reclaiming. We then detail MemGuard's reservation and reclaiming mechanisms that manage the guaranteed bandwidth to provide efficient performance isolation between cores that share the memory subsystem.

3.3.1 Guaranteed Memory Bandwidth

Figure 3.5 shows the organization of a typical DRAM based memory system. A DRAM module is composed of several DRAM chips that are connected in parallel to form a wide interface (64bits I/F for DDR). Each DRAM chip has multiple banks that can be operated concurrently. Each bank is then organized as a 2d array consisting with rows and columns. A location in DRAM can be addressed with the bank, row and column number.

In each bank, there is a buffer, called *row buffer*, to store a single row (typically 1$\tilde{2}$KB) in the bank. In order to access data, the DRAM controller must first copy the row containing the data into the row buffer (i.e., opening a row). The required latency for this operation is denoted as $tRCD$ in DRAM specifications. The DRAM controller then can read/write from the row buffer with only issuing column addresses, as long as the requested data is in the

```
1  function periodic_timer_handler ;
2  begin
3      Q_i^predict ← output of usage predictor ;
4      Q_i ← user assigned static budget ;
5      if Q_i^predict > Q_i then
6          q_i ← Q_i;
7      else
8          q_i ← Q_i^predict ;
9      G += max{0, Q_i − q_i};
10     program PMC to cause overflow interrupt at q_i;
11     re-schedule all dequeued tasks;
12 function overflow_interrupt_handler ;
13 begin
14     u_i ← used budget in the current period ;
15     if G > 0 then
16         if u_i < Q_i then
17             q_i ← min{Q_i − u_i, G} ;
18         else
19             q_i ← min{Q_min, G} ;
20         G -= q_i;
21         program PMC to cause overflow interrupt at q_i ;
22         Return ;
23     if u_i < Q_i then
24         Return ;
25     if ∑ u_i = r_min then
26         manage best-effort bandwidth (see Section 3.4) ;
27     de-schedule tasks in the CPU run-queue ;
```

Figure 3.4: MemGuard Implementation

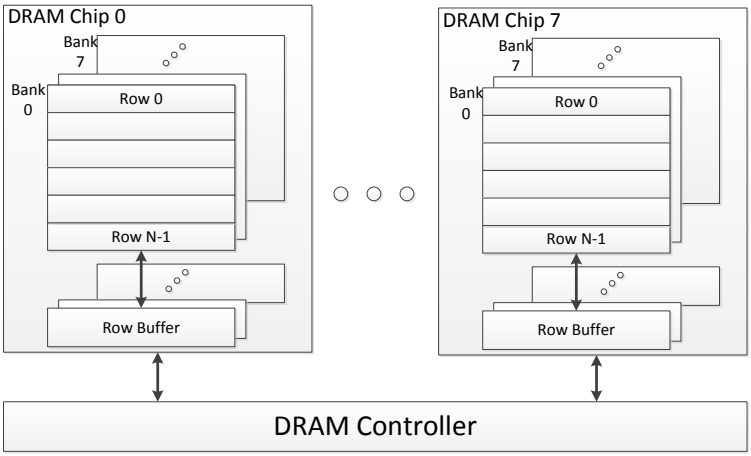


Figure 3.5: DRAM based memory system organization—Adopted from Fig. 2 in [1]

same row. The associated latency is denoted as tCL. If the requested data is in a different row, however, it must save the content of the row buffer back to the originating row (i.e., closing a row) and then open the new row. The associated latency is denoted as tRP. The access latency to a memory location, therefore, varies depending on whether the data is already in the row buffer (i.e., row hit) or not (i.e., row miss). Moreover, because banks can be accessed in parallel, the achievable memory bandwidth also heavily depends on how the banks are accessed: For example, if two successive memory requests are targeting to bank0 and bank1 respectively, they can be requested in parallel, achieving higher bandwidth.

Considering these characteristics, we can distinguish the maximum (peak) bandwidth and the guaranteed (worst-case) bandwidth as follows:

- **Maximum (peak) bandwidth:** This is stated maximum performance of a given memory module where the speed is only limited by I/O bus clock speed. For example, a PC6400 DDR2 memory module's peak bandwidth is 6400MB/s where it can transfer 64bit data at the rising and falling edge of bus cycle running at 400MHz.

- **Guaranteed (worst-case) bandwidth:** Achieved bandwidth is, however, limited by available parallelism and various timing requirements. The worst-case occurs when all memory requests are targeting to a single memory bank (bank conflict), and each successive request is ac-

cessing different row, causing a row switch (row miss). This can be calculated from the DRAM specification using tRC parameter.

Because the guaranteed bandwidth can be satisfied regardless of memory access locations, we use the guaranteed bandwidth as the basis for bandwidth reservation and reclaiming as we will detail in the subsequent subsections.

3.3.2 Memory Bandwidth Reservation

MemGuard provides two levels of memory bandwidth reservation: system-wide reservation and per-core reservation.

1. System-wide reservation regulates the total allowed memory bandwidth such that it does not exceed the guaranteed bandwidth — denoted as r_{min}.

2. Per-core reservation assigns a fraction of r_{min} to each core, hence each core reserves bandwidth B_i and $r_{min} = \sum_{i=0}^{m} B_i$.

Each regulator reserves memory bandwidth represented by memory access budget Q_i for every period P: i.e. $B_i = \frac{Q_i}{P}$. Regulation period P is a system-wide parameter and should be small to effectively enforce specified memory bandwidth. Although small regulation period is better for predictability, there is a practical limit on reducing the period due to interrupt and scheduling overhead; we currently configure the period as 1ms. The reservation follows the common resource reservation rules [88, 98, 36]. Per-core instant budget q_i is deducted as the core consumes memory bandwidth. For accurately accounting memory usage, we use per-core PMC interrupt: specifically, we program the PMC at the beginning of each regulation period so that it generates an interrupt when q_i is depleted for Core i. Once the interrupt is received, the regulator calls the OS scheduler to schedule a high-priority real-time task to effectively idle the core until the next period begins. At the beginning of the next period the budget is replenished in full and the real-time "idle" task will be suspended so that regular tasks can be scheduled.

3.3.3 Memory Bandwidth Reclaiming

Each core has statically assigned bandwidth Q_i as the baseline. It also maintains an instant budget q_i to actually program the PMC which can vary at each period based the output of the memory usage predictor. We currently use an Exponentially Weighted Moving Average (EWMA) filter as the memory usage predictor which takes the memory bandwidth usage of the previous period as input. The reclaim manager maintains a global shared budget G. It collects surplus bandwidth from each core and re-distributes it when in need. Note that G is initialized at the beginning of each period and any unused G is discarded at the end of this period. Each core only communicates with the central reclaim manager for donating and reclaiming its budget. This avoids possible circular reclaiming among all cores and greatly reduces implementation complexity and runtime overhead.

The details of the reclaiming rules are as follows:

1. At the beginning of each regulation period, the current per-core budget q_i is updated as follows:

   $$q_i = \min\{Q_i^{predict}, Q_i\}$$

   If the core is *predicted* not to use the full amount of the assigned budget Q_i, the current budget is set the predicted usage, $Q_i^{predict}$ (See Line 5-8 in Figure 3.4).

2. At the beginning of each regulation period, the global budget G is updated as follows:

   $$G = \sum_i \{Q_i - q_i\}.$$

   Each core donates its spare budget to G (See Line 9 in Figure 3.4).

3. During execution, the core can reclaim from the global budget if its corresponding budget is depleted. The amount of reclaim depends on the requesting core's condition: If the core's used budget u_i is less than the static reserved bandwidth Q_i (this happens when the prediction is smaller than Q_i), then it tries to reclaim amount equal to the difference between Q_i and the current usage u_i; if the core used equal or greater than the assigned budget Q_i, it only gets up to a small fixed amount of

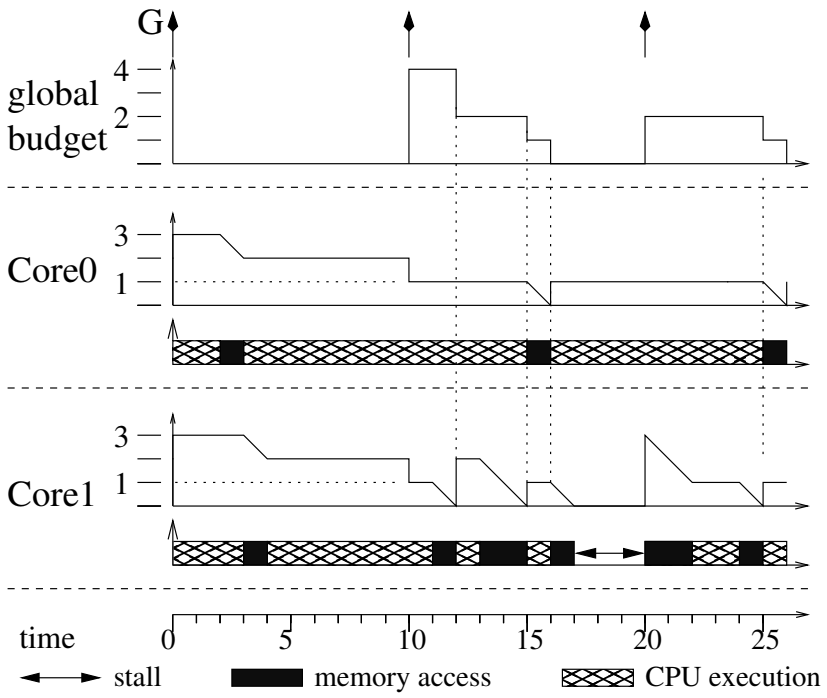


Figure 3.6: An illustrative example with two cores

budget, Q_{min} (See Line 16-19 in Figure 3.4). If Q_{min} is too small, too many interrupts can be generated within a period, increasing overhead. As such, it is a configuration parameter and empirically determined for each particular system.

4. Since our reclaim algorithm is based on prediction, it is possible a core may not be able to use the originally assigned budget Q_i. This can happen when the core donates its budget too much (due to mis-prediction) and other cores already reclaimed the entire donated budget (i.e., $G = 0$) before the core tries to reclaim. When this happens, our current heuristic is to allow the core continue execution, hoping that it may use its Q_i, although it is not guaranteed (See Line 23-24 in Figure 3.4). At the beginning of the next period, we verify if it was able to use the budget. If not, we call it a *reclaim underrun error* and notify the predictor of the difference $(Q_i - u_i)$. The predictor then tries to *compensate* it by using $Q_i + (Q_i - u_i)$ as its input for the next period.

3.3.4 Example

Figure 3.6 shows an example with two cores, each with an assigned static budget 3 (i.e., $Q_0 = Q_1 = 3$). The regulation period is 10 time units and the arrows at the top of the figure represent the period activation times. The figure demonstrates the global budget together with these two cores.

When the system starts, each core starts with the assigned budget 3. At time 10, the prediction for each core is 1 as it only used budget 1 within the period [0,10], hence, the instant budget becomes 1 and the global budget G becomes 4 (each core donates 2). At time 12, Core 1 depletes its instant budget. Since its assigned budget is 3, Core 1 tries to reclaim 2 from G and G becomes 2. At time 15, Core 1 depletes its budget again. This time Core 1 already used its assigned budget, only a fixed amount of extra budget (Q_{min}) 1 is reclaimed from G and G becomes 1. At time 16, Core 0 depletes its budget. Since G is 1 at this point, Core 0 only reclaims 1 and G drops to 0. At time 17, Core 1 depletes its budget again then it dequeues all the tasks as it cannot reclaim additional budget from G. When the third period starts at time 20, the $Q_1^{predict}$ is larger than Q_1. Therefore, Core 1 gets the full amount of assigned budget 3, according to Rule 1 in Section 3.3.3, while Core 0 only gets 1, and donates 2 to G. At time 25, after Core 1 depletes its budget, Core 1 reclaims an additional budget Q_{min} from G.

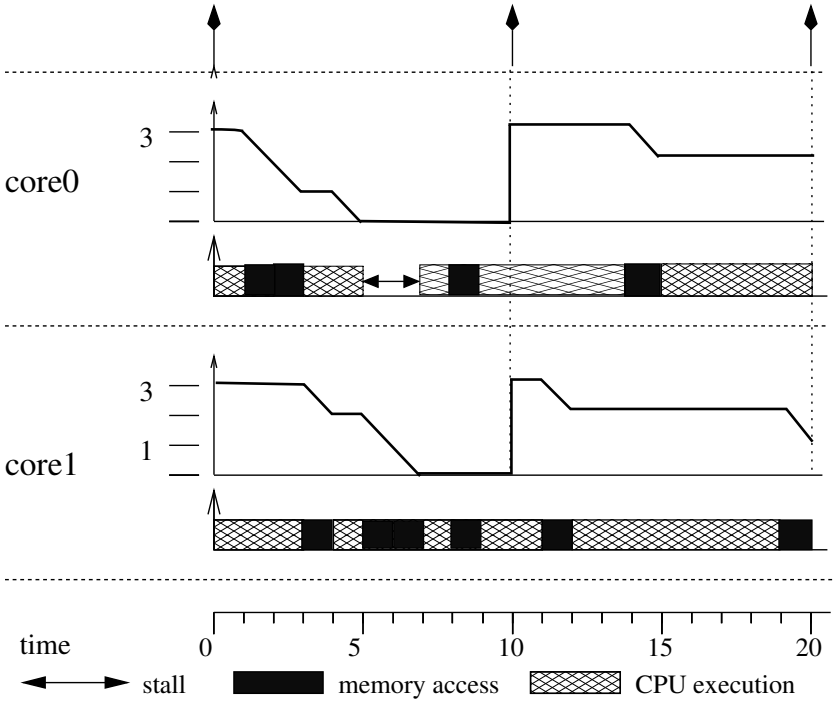
3.4 Best-effort Bandwidth Management

In this section, we first define the best-effort memory bandwidth in MemGuard and describe two proposed management schemes: spare sharing and propotional sharing.

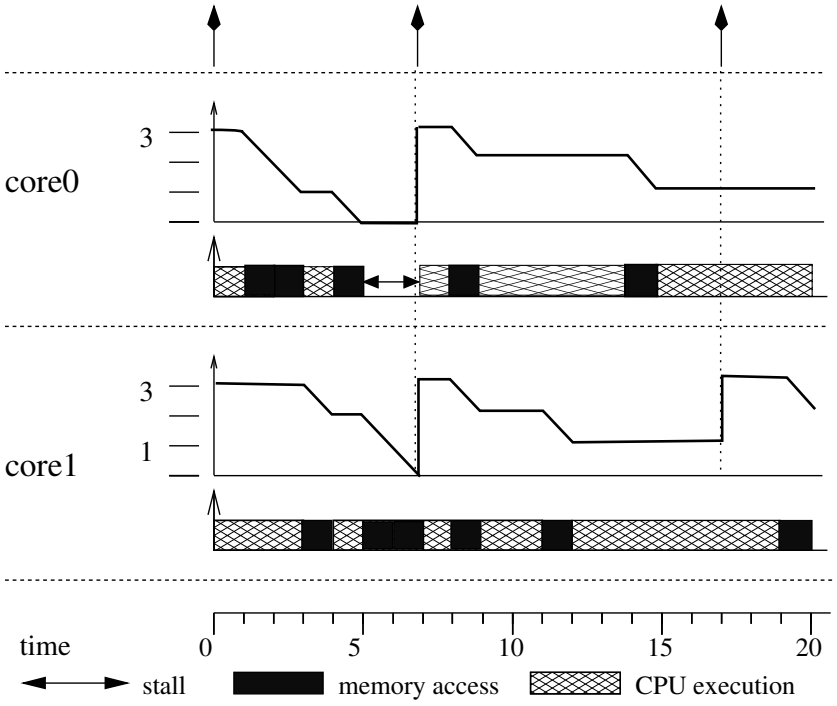
3.4.1 Best-effort Memory Bandwidth

We define best-effort memory bandwidth as any additional achieved bandwidth above the guaranteed bandwidth r_{min}. Because r_{min} is smaller than the peak bandwidth, efficient utilization of best-effort bandwidth is important, especially for memory intensive tasks.

As described in the previous section, memory reservation and reclaiming



(a) Spare Sharing



(b) Proportional Sharing

Figure 3.7: Comparison of best-effort bandwidth management schemes

works w.r.t. r_{min} in MemGuard. If all cores exhaust their given bandwidths before the current period ends, all cores would wait for the next period doing nothing under the rules described in Section 3.3. Since MemGuard already delivered reserved bandwidth to each core, any additional bandwidth is now considered as best-effort bandwidth. We propose two best-effort bandwidth management schemes in the following subsections.

3.4.2 Spare Sharing

When all cores collectively use their assigned budgets (Line 25 in Figure 3.4), the spare sharing scheme simply let all cores compete the memory until the next period begins. This strategy maximize throughput as it effectively equivalent as temporarily disabling MemGuard. In another perspective, it gives an equal chance for each core to utilize the remaining best-effort bandwidth, regardless of its reserved memory bandwidth.

Figure 3.7(a) shows an example operation of the scheme. At time 5, Core 0 depletes its budget and it dequeues all tasks in the core, assuming the G is zero in the period. At time 7, Core 1 depletes its budget. At this point, there are no cores that have remaining budgets. Therefore, Core 1 sends a broadcast message to wake-up the Core 0 and both cores compete the memory until the next period begins at time 10.

Note that this mechanism only starts *after* all the cores have depleted their assigned budgets. The reason is that if a core has not yet used q_i, allowing other cores to execute may bring intensive memory contention, preventing the core from using the remaining q_i.

3.4.3 Proportional Sharing

The proportional sharing scheme also starts after all cores use their budgets like the spare-sharing scheme, described in the previous subsection but it differs in that it starts a new period immediately instead of waiting the remaining time in the period. This effectively makes each core to utilize the best-effort bandwidth proportional to its reserved bandwidth—i.e., the more the q_i, the more best-effort bandwidth the core gets.

Figure 3.7(b) shows an example operation of the scheme. At time 7, when

all cores use their budgets, it start a new period and each core's budget is recharged immediately. This means that the length of each period can be shorter depending on workload. However, if the condition is not met, i.e., there is at least one core that does not use its budget, then the same fixed period length is enforced.

This scheme bears some similarities with the IRIS algorithm, a CPU bandwidth reclaiming algorithm [92], that extends CBS [37] to solve the deadline aging problem of CBS; In the original CBS, when a server exhaust its budget, it immediately recharges the budget and extend the deadline. This can cause a very long delay to CPU intensive servers that extend deadlines very far ahead while other servers are inactive. The IRIS algorithm solves this problem by introducing a notion of recharging time in which a server that exhausts its budget must wait until it reaches the recharging time. If there is no active server and there is at least one server that wait for recharging, IRIS update server's time to the earliest recharging time. This is similar to the proportional sharing scheme presented in this subsection in that servers' budgets are recharged when all servers use their given budgets. They are, however, significantly different in the sense that proportional sharing is designed for sharing memory bandwidth between multiple concurrently accessing cores and is it is not based on CBS scheduling method.

3.5 Evaluation Setup

In this section, we introduce the platform used for the evaluation and the software system implementation.

3.5.1 Evaluation platform

Figure 4.4 shows the architecture of our testbed, an Intel Core2Quad Q8400 processor. The processor is clocked at 2.66GHz and has four physical cores. It contains two separate 2MB L2 caches; each L2 cache is shared between two cores.

As our focus is not the shared cache, we use cores that do not share the same LLC for experiments (i.e., Core 0 and Core 2 in Figure 4.4). We do, however, use four cores when tasks are not sensitive to shared LLC by nature

(e.g., working set size of each task is bigger than the LLC size).

We use two 2GB PC6400 DRAM modules in a dual-channel configuration, each of which is having two ranks and 8 banks (8 x 14 x 10 x 64). We emphirically estimated the guaranteed bandwidth r_{min} of the memory subsystem as 1.2GB, which is used to configure the MemGuard in the rest of our evaluation. Note that this number may vary depending on the number of memory banks and other timing parameters. The smaller the r_{min} is the better r_{min} can be guaranteed at the cost of less reservable memory bandwidth and low overall memory bandwidth utilization. As shown in Section 3.6.1, the estimated r_{min} provides strong isolation performance in our experiments.

In order to account per-core memory bandwidth usage, we used a LLC miss performance counter [2] per each core. Since the LLC miss counter does not account prefetched memory traffic, we disabled all hardware prefetchers [3].

Note that LLC miss counts do not capture LLC write-back traffic which may underestimate actual memory traffic, particularly for write-heavy benchmarks. However, because SPEC2006 benchmarks, which we used in evaluation, are read heavy (only 20% of memory references are write [99]); and memory controllers often implement write-buffers that can be flushed later in time (e.g., when DRAM is not in use by other outstanding read requests) and writes are considered to be completed when they are written to the buffers [100], write-back traffic do not necessarily cause additional latency in accessing DRAM. Analyzing the impact of accounting write-back traffic in terms of performance isolation and throughput is left as future work.

3.5.2 Software Implementation

We implemented MemGuard in Linux version 3.6 as a kernel module [4]. We use the *perf_event* infrastructure to install the counter overflow handler at each period. The logic of both handlers is shown in Figure 3.4.

Compared to our previous implementation [6], we made several changes. First, as explained in Section 3.3, we now schedule a high-priority real-time task to "throttle" the core instead of de-scheduling all the tasks in the core.

[2] LAST_LEVEL_CACHE_MISSES: event=2e, umaks=41. See [46]

[3] We used `http://www.eece.maine.edu/~vweaver/projects/prefetch-disable/`

[4] MemGuard is available at https://github.com/airtight/memguard

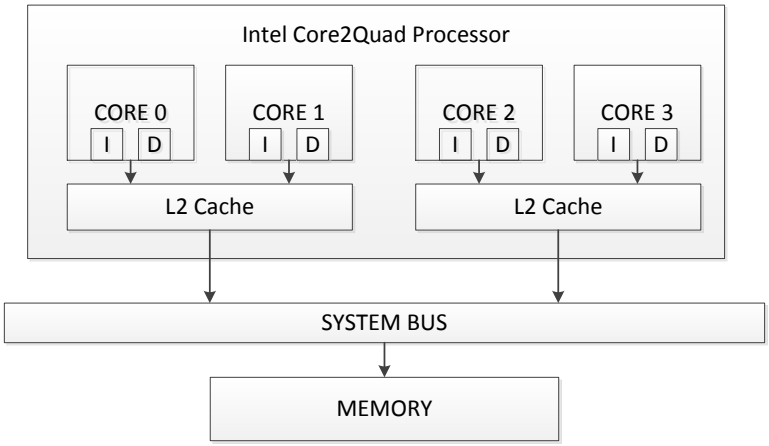


Figure 3.8: Hardware architecture of our evaluation platform.

This solves the problem of linearly increasing overhead as the number of tasks increase. Second, we now use a separate timer for pacing the regulation period, instead of using the OS tick, in order to easily re-program the length of period. It is needed to support proportional sharing mode which may change the period length on-demand.

MemGuard supports several memory bandwidth reservation modes, namely per-core bandwidth assignment and per-task assignment mode. In per-core mode, system designers can assign absolute bandwidth (e.g., 200MB/s) or relative weight expressing relative importance. In the latter case, the actual bandwidth is calculated at every period by checking active cores (cores that have runnable tasks in their runqueues). In per-task mode, the task priority is used as weight for the core the task runs on. Notice that in this mode, a task can migrate to a different core with its own memory bandwidth reservation. In Section 3.6, we use per-core assignment mode using both absolute bandwidth and weight depending on experiments.

3.6 Evaluation Results and Analysis

In this section, we evaluate MemGuard in terms of performance isolation guarantee and throughput with a set of synthetic and SPEC2006 benchmarks.

For evaluation, we use four different modes of MemGuard: reserve only (MemGuard-RO), b/w reclaim (MemGuard-BR), b/w reclaim + spare share

(MemGuard-BR+SS), and b/w reclaim + proportional sharing (MemGuard-BR+PS). In MemGuard-RO mode, each core only can use its reserved b/w as described in Section 3.3.2. The MemGuard-BR mode uses the predictive memory bandwidth reclaiming algorithm described in Section 3.3.3. Both MemGuard-BR+SS mode and MemGuard-BR+PS use the bandwidth reclaiming but differ in how to manage the best-effort bandwidth after all cores collectively consume the guaranteed bandwidth as described in Section 3.4.2 and Section 3.4.3 respectively.

3.6.1 Isolation Effect of Reservation

In this experiment, we illustrate the effect of memory bandwidth reservation on performance isolation by configuring MemGuard with the reservation only mode (MemGuard-RO). We pair the most memory intensive benchmark, 470.lbm as the background task, with a foreground task selected from SPEC2006 benchmarks. Each foreground task runs on Core 0 with 1.0GB/s memory bandwidth reservation while the background task runs on Core 2 with reservation varying from 0.2GB/s to 2.0GB/s. Note that assigning more than 0.2GB/s on Core 2 makes the total bandwidth exceeds the estimated minimum DRAM service rate of 1.2GB/s. Note that MemGuard-RO mode only allows each core to use its assigned bandwidth only regardless of memory activities in other cores.

Figure 3.9 shows the IPC of each foreground task, normalized to the IPC measured in isolation (i.e., no background task) with the same 1.0GB/s reservation. First, notice that when we assign 0.2GB/s to Core 2 (denoted "w/ lbm:0.2G") the IPC of each task is very close to the ideal value 1.0—i.e., negligible performance impact from the co-running background task. However, as we increase the assigned memory bandwidth of Core 2, the IPC of the foreground task gradually decreases below 1.0—i.e., performance isolation is violated due to increased memory contention. For example, 462.libquantum on Core0 shows 30% IPC reduction when the background task is running on Core2 with 2.0GB/s reservation (denoted "w/ lbm:2.0G").

These results demonstrate that performance isolation can be achieved by regulating the aggregated total request rate. Specifically, limiting the rate (to be smaller than r_{min}) achieves performance isolation for the SPEC bench-

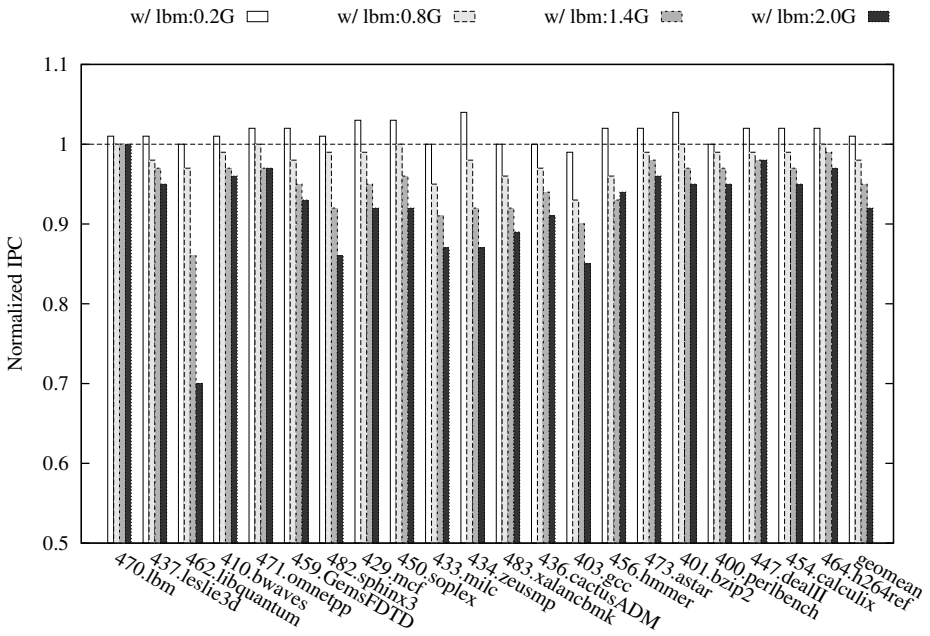


Figure 3.9: Normalized IPC of a subset of SPEC2006 (Core 0), co-scheduled with 470.lbm (Core 2)

marks shown in this figure. The rest of SPEC benchmarks also show consistent behavior but we omit them as they show less interference (less slowdown).

3.6.2 Results with SPEC2006 Benchmarks on Two Cores

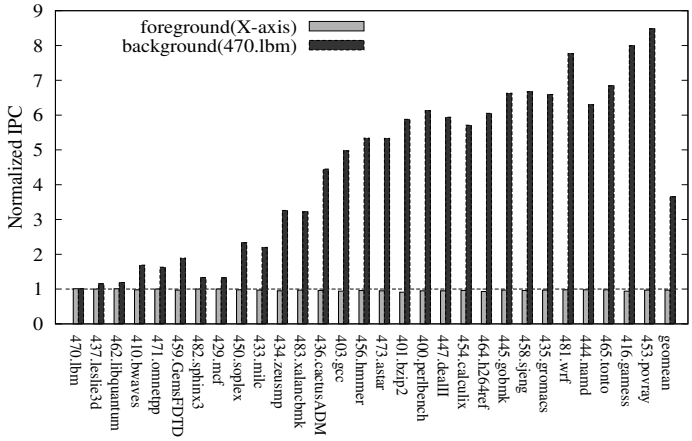
We now evaluate MemGuard using the entire SPEC 2006 benchmarks. We first profile the benchmarks to better understand of their characteristics. We run each benchmark for 10 seconds with the reference input and measure the instruction counts and LLC miss counts, using *perf* tool included in the Linux kernel source tree, to calculate the average IPC and the memory bandwidth usage. We multiply the LLC miss count with the cache-line size (64 bytes in our testbed) to get the total memory bandwidth usage.

Table 3.1 shows the characteristics of each SPEC2006 benchmark, in decreasing order of average memory bandwidth usage, when each benchmark runs alone on our evaluation platform. Notice that the benchmarks cover a wide range of memory bandwidth usage, ranging from 1MB/s (453.povray) up to 2.1GB/s (470.lbm).

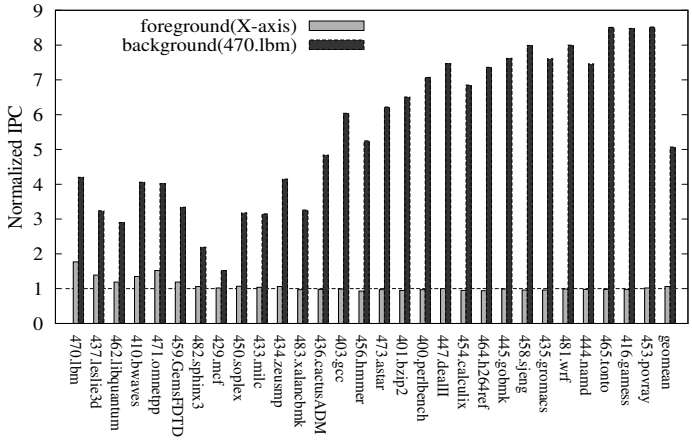
We now evaluate the effect of the b/w reclaim algorithm (MemGuard-BR)

Benchmark	Avg. IPC	Avg. B/W(MB/s)	Memory Intensity
470.lbm	0.52	2121	
437.leslie3d	0.51	1581	
462.libquantum	0.60	1543	
410.bwaves	0.62	1485	
471.omnetpp	0.83	1373	High
459.GemsFDTD	0.50	1203	
482.sphinx3	0.58	1181	
429.mcf	0.18	1076	
450.soplex	0.54	1025	
433.milc	0.59	989	
434.zeusmp	0.93	808	
483.xalancbmk	0.54	681	
436.cactusADM	0.68	562	
403.gcc	0.98	419	
456.hmmer	1.53	317	Medium
473.astar	0.58	307	
401.bzip2	0.97	221	
400.perlbench	1.36	120	
447.dealII	1.41	118	
454.calculix	1.53	113	
464.h264ref	1.42	101	
445.gobmk	0.97	95	
458.sjeng	1.10	74	
435.gromacs	0.86	60	
481.wrf	1.73	38	Low
444.namd	1.47	18	
465.tonto	1.38	2	
416.gamess	1.34	1	
453.povray	1.17	1	

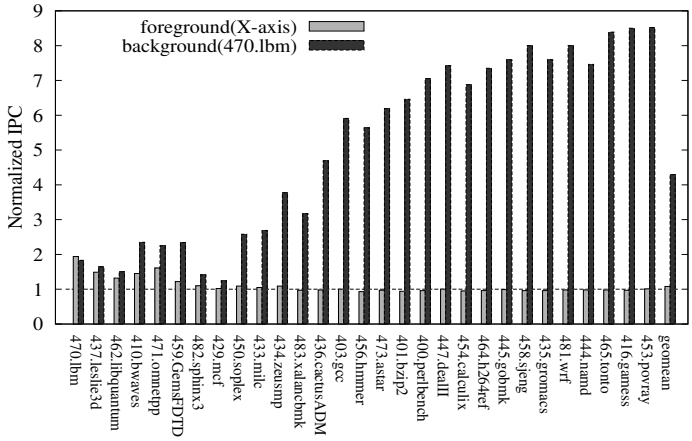
Table 3.1: SPEC2006 characteristics



(a) MemGuard-BR (b/w reclaim)

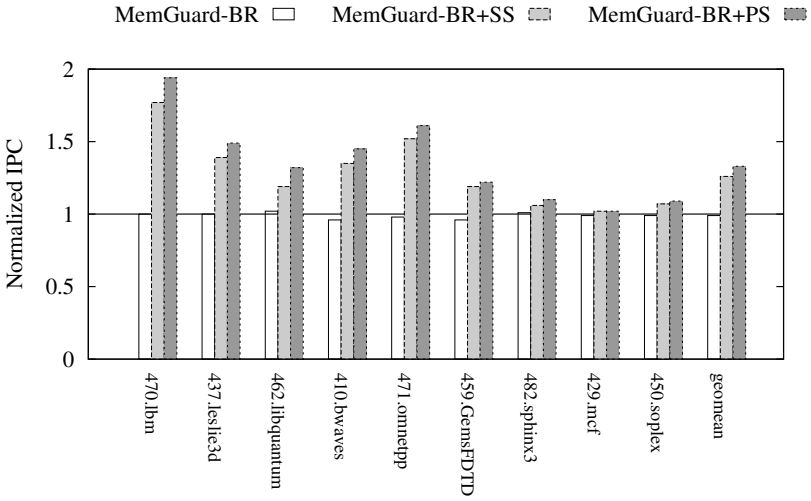


(b) MemGuard-BR+SS (b/w reclaim + spare share)

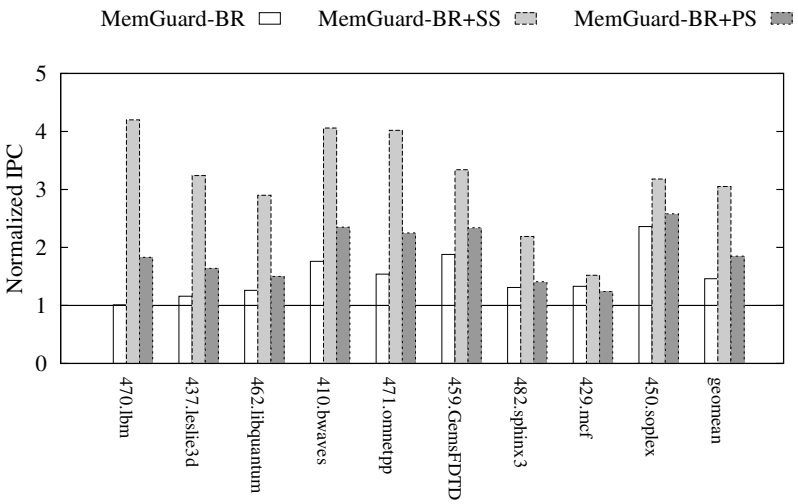


(c) MemGuard-BR+PS (b/w reclaim + proportional share)

Figure 3.10: Normalized IPCs of co-scheduled SPEC2006 benchmarks.



(a) Foreground (X-axis)@Core0



(b) Background (470.lbm)@Core2

Figure 3.11: Normalized IPC of nine memory intensive SPEC2006 benchmarks (a) and the co-running 470.lbm (b). The X-axis shows the foreground task on Core 0 in both sub-figures.

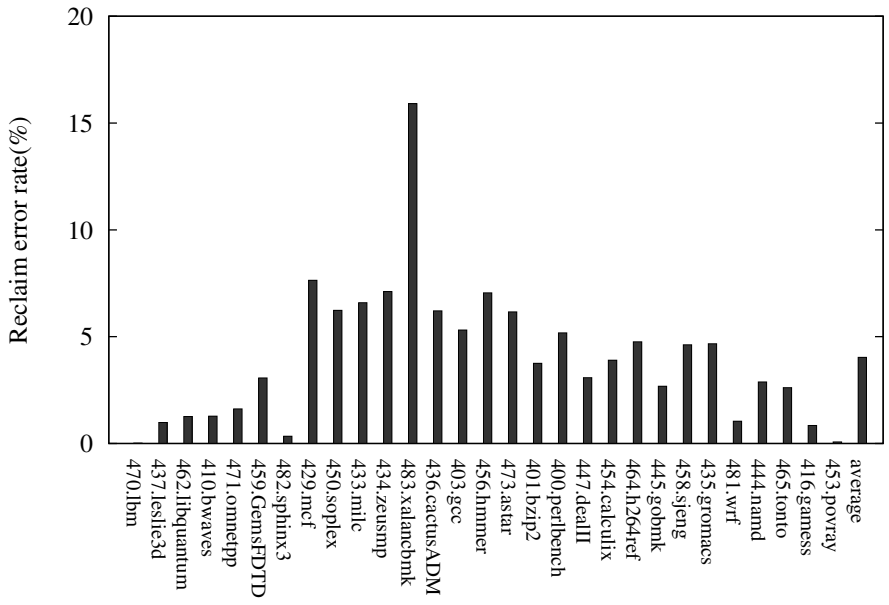


Figure 3.12: Reclaim underrun error rate when using bandwidth reclaim mode (MemGuard-BR)

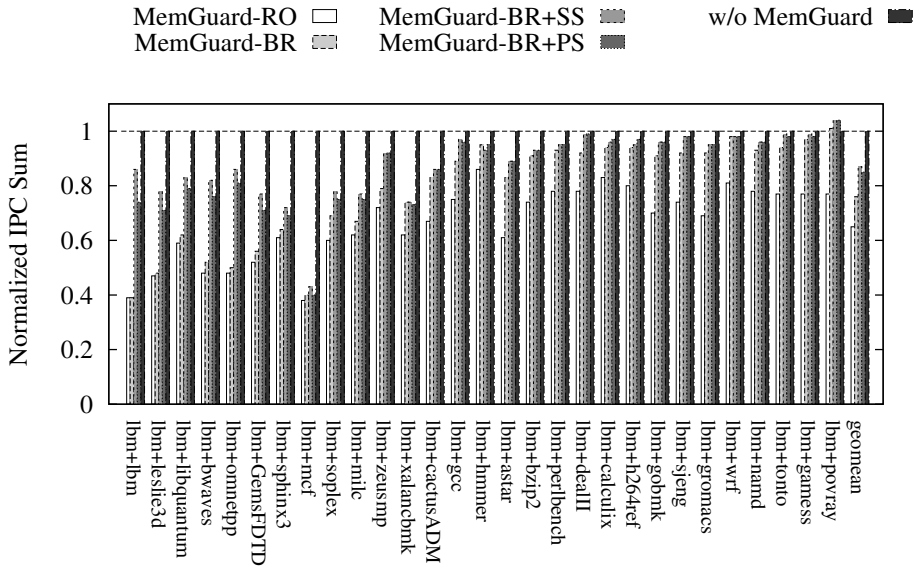


Figure 3.13: Normalized IPC sum (throughput)

and two best-effort bandwidth management schemes (MemGuard-BR+SS and MemGuard-BR+PS) on a dual-core system configuration.

Figure 3.10 shows the normalized IPCs (w.r.t. MemGuard-RO) of co-scheduled foreground and background task using MemGuard-RO, MemGuard-BR+SS, and MemGuard-BR+PS. The foreground tasks in X-axis are sorted in decreasing order of memory intensity. For all task pairs, the foreground task runs on Core0 with 1.0GB/s reservation and the background task (470.lbm) run on Core2 with 0.2GB/s reservation. Notice that the Core2 is severely under-reserved as 470.lbm's average bandwidth is above 2GB/s (see Table 3.1).

MemGuard-BR shows the effect of our bandwidth reclaiming algorithm. For most pairs, the background task achieves a higher IPC compared to the baseline (i.e., MemGuard-RO). This can be explained as follows: if a foreground task does not use the assigned budget, the corresponding background task can effectively reclaim the unused budget and make more progress. In particular, the background tasks in the right side of the figure (from 433.milc on the X-axis) show significant performance improvements. This is because the corresponding foreground tasks use considerably smaller average bandwidth than their assigned budgets. Consequently, background tasks can reclaim more budgets and achieve higher performance. The average IPC of all background tasks is improved by 3.7x, compared to the baseline, showing the effectiveness of the reclaiming algorithm.

Note that the slowdown of foreground task, due to reclaiming of background task, is small—less than 3% on average. The slight performance reduction, i.e., reduced performance isolation, can be considered as a limitation of our prediction based approach that can result in reclaim underrun error as described in Section 3.3.3. To better understand this, Figure 3.12 shows reclaim underrun error rates (error periods / total periods) of the experiments used to draw Figure 3.10(a). On average, the error rate is 4% and the worst case error rate is 16% for 483.xalancbmk. Note that although 483.xalancbmk suffers higher reclaim underrun error rate, it does not suffer noticeable performance degradation because the absolute difference between the reserved bandwidth and the achieved bandwidth is relatively small in most periods that suffered reclaim underrun errors.

MemGuard-BR+SS enables the spare bandwidth sharing algorithm (Section 3.4.2) on top of MemGuard-BR. Compared to Figure 3.10(a), the tasks

in the left side of the figure—i.e., task pairs coupled with more memory intensive foreground tasks—show noticeable improvements. This is because that after both tasks (the foreground and the background) collectively consume the total reserved bandwidth (r_{min}), the spare bandwidth sharing mode allows both tasks to continue until the beginning of the next period, making more progress on both tasks. On average, the performance is improved by 5.1x for background tasks and by 1.06x for foreground tasks, compared to the baseline.

MemGuard-BR+PS enables the proportional sharing mode (Section 3.4.3) on top of MemGuard-BR. While it also improves performance of both foreground and background tasks as in MemGuard-BR+SS, the average improvement of background tasks is only 4.3x, compared to 5.1x in MemGuard-BR+SS. On the other hand, the average improvement of foreground tasks is 1.08x, compared to 1.06x in the MemGuard-BR+SS mode. This is because the proportional sharing mode provides much less bandwidth to the background tasks as it begins a new period immediately after the guaranteed bandwidth is consumed, while the spare sharing mode let the background task freely competes with the foreground task until the next period begins, hence achieves more bandwidth.

The differences of the two modes—MemGuard-BR+SS and MemGuard-BR+PS—can be seen more clearly by investigating the "High" memory intensity foreground tasks and the corresponding background tasks separately as shown in Figure 3.11. In all cases, the proportional sharing mode improves foreground tasks' performance at the cost of reduced background tasks' performance. Hence, the proportional sharing mode is useful when we want to prioritize certain cores with more guaranteed bandwidth reservations over other cores with less reservations.

Figure 3.13 compares throughput of four MemGuard modes (MemGuard-RO, MemGuard-BR, MemGuard-BR+SS, and MemGuard-BR+PS) and the vanilla kernel without using MemGuard (Vanilla.) Here we define throughput simply as the sum of IPCs of each foreground and background task pair. The Y-axis shows the normalized IPC sum (w.r.t. Vanilla) of each pair of foreground and background tasks that represents the system throughput of the pair. Compared to MemGuard-RO, MemGuard-BR achieves 11% more throughput on average (geometric mean). Both MemGuard-BR+SS and MemGuard-BR+PS achieve additional 11% and 9% improvement re-

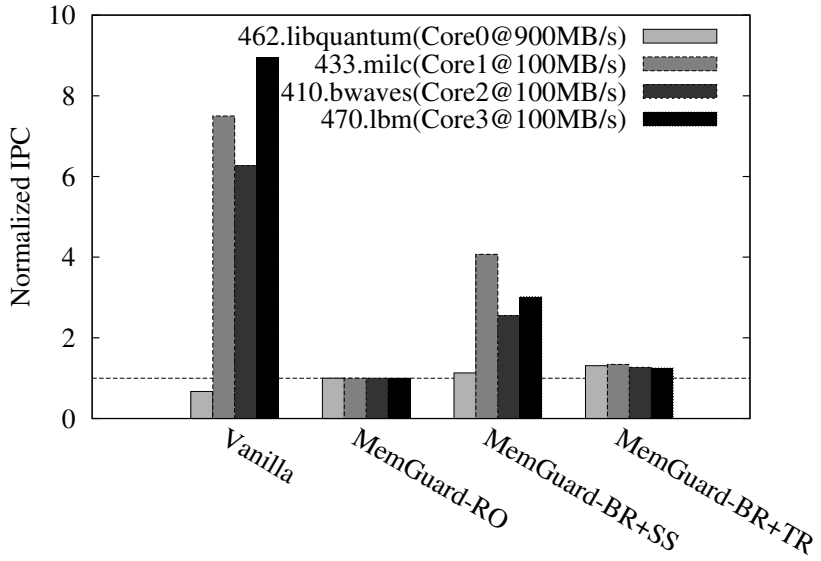
spectively. Although Vanilla achieves higher throughput in general, it does not provide performance isolation while MemGuard provides performance isolation at a reasonable throughput cost.

3.6.3 Results with SPEC2006 on Four Cores

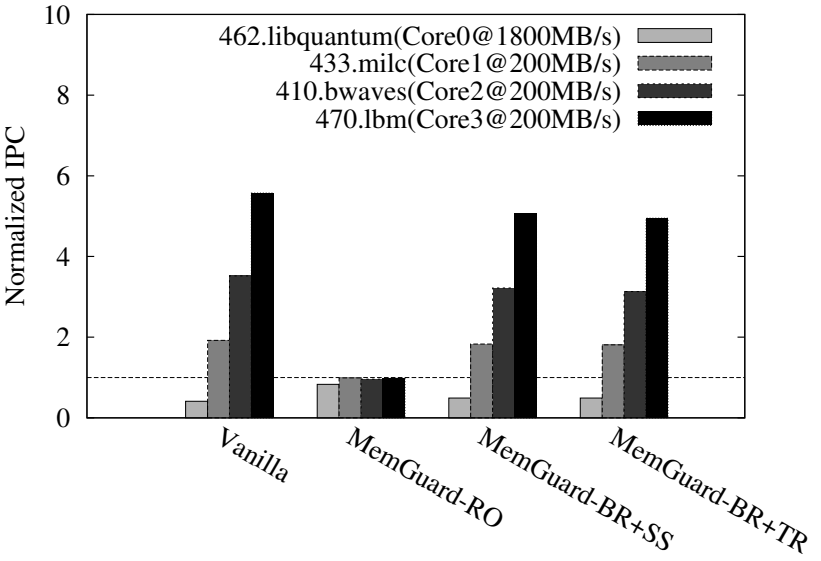
In this experiment, we evaluate MemGuard using all four cores in our testbed. We use four SPEC benchmarks—462.libquantum, 433.milc, 410.bwaves, and 470.lbm—each of which runs on one core in the system.

Because the testbed has two shared LLC caches, each of which is shared by two cores, we carefully choose the benchmarks in order to minimize cache storage interference effect. To this end, we experimentally verify each benchmark by running it together with one synthetic cache trash task in both shared and separate LLC configurations; if performance of the two configurations differ less than 5%, we categorize the benchmark as LLC insensitive.

Figure 3.14(a) shows the normalized IPC (w.r.t. MemGuard-RO where each task is scheduled in isolation) of each task when all four tasks are co-scheduled using MemGuard in three different modes (MemGuard-RO, MemGuard-BR+SS, and MemGuard-BR+PS) and without using MemGuard (Vanilla). The weight assignment is 9:1:1:1 (for 462.libquantum, 433.milc, 410.bwaves, and 470.lbm respectively) and the r_{min} is 1.2GB/s. Vanilla is unaware of the weight assignment. Hence, the high-priority 462.libquantum on Core 0 is 33% slower than the baseline reservation due to contentions from other low priority tasks. Although it is clear that overall throughput is higher in Vanilla, it cannot provide isolated performance guarantee for one specific task, in this case 462.libquantum. In contrast, MemGuard-RO delivers exactly the performance that is promised by each task's reserved bandwidth (i.e., baseline) without experiencing noticeable slowdowns by interferences from co-running tasks. Hence it can guarantee performance of the high-priority task (462.libquantum) at the cost of significant slowdowns of low-priority tasks. MemGuard-BR+SS improves performance of all tasks beyond their guaranteed performances by sharing best-effort bandwidth—through bandwidth reclaiming and spare sharing. This is especially effective for low-priority tasks as they are improved by 4.07x, 2.55x, 3.02x (433.milc, 410.bwaves, 470.lbm respectively) compared to the baseline. The



(a) r_{min}=1.2GB/s



(b) r_{min}=2.4GB/s

Figure 3.14: Isolation and throughput impact of r_{min}.

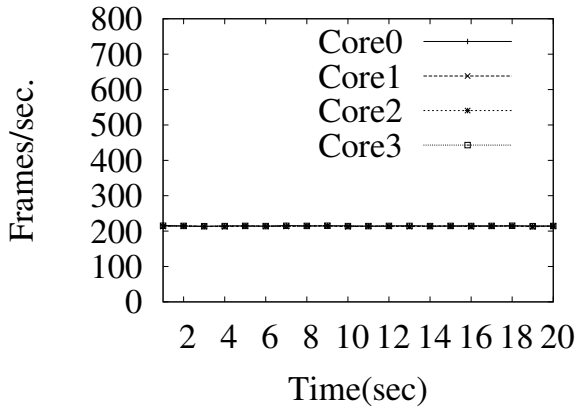
high-priority task (462.libquantum) is also improved by 1.13x. MemGuard-BR+PS also improves performance of all tasks above their guaranteed performances, but different in that it favors the high-priority task over low-priority tasks: the high-priority task is improved by 1.31x while low-priority tasks are improved by 1.34x, 1.27x, and 1.25x. This is because MemGuard-BR+PS enforces the assigned weight all the time, by starting a new period immediately when the guaranteed bandwidth is used, while MemGuard-BR+SS doesn't between the time it satisfies the guaranteed bandwidth and the time when the next period starts (the interval is fixed in MemGuard-BR+SS).

Figure 3.14(b) follows the same weight settings but doubles the r_{min} value to 2.4GB/s in order to compare its effect on throughput and performance isolation. Because the r_{min} is doubled, each core's reserved bandwidth is doubled and the baseline of each task (Y-axis value of 1) is changed accordingly. Note first that MemGuard-RO does not guarantee performance isolation anymore as 462.libquantum is 17% slower than the baseline. It is because the 2.4GB/s bandwidth can not be guaranteed by the given memory system, causing additional queuing delay to the 462.libquantum. This is consistent with our finding in Section 3.6.1. In both MemGuard-BR+SS and MemGuard-BR+PS, the IPC of 462.libquantum is further reduced, because other cores can generate more interference using reclaimed bandwidth that 462.libquantum donated. On the other hand, both modes achieve higher overall throughput as they behave very similar to Vanilla. This shows the trade-off between throughput and performance isolation when using MemGuard.

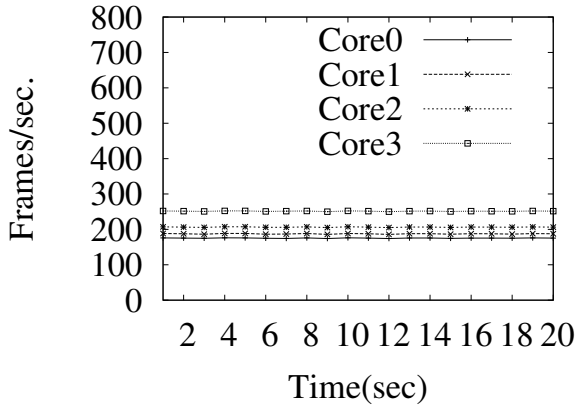
3.6.4 Effect on Soft Real-time Applications

We illustrate the effect of MemGuard for soft real-time applications using a synthetic soft real-time image processing benchmark *fps*. The benchmark processes an array of two HD images (each image is 32bpp HD data: 1920x1080x4 bytes = 7.9MB) in sequence. It is greedy in the sense that it attempts to process as quickly as possible.

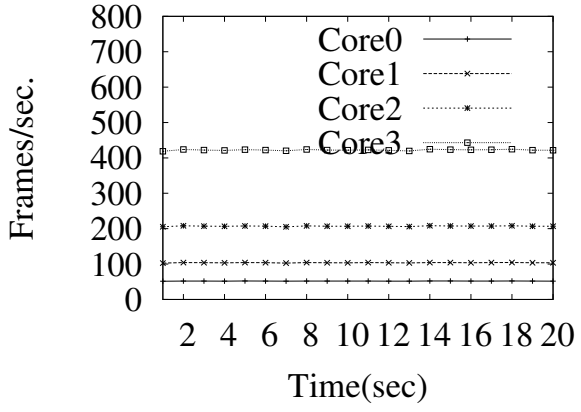
Figure 3.14(a) shows frame-rates of *fps* instances on our 4-core system using MemGuard in two different modes (MemGuard-BR+SS, MemGuard-BR+PS) and without using MemGuard (Vanilla). The weight assignment



(a) Vanilla



(b) MemGuard-BR+SS



(c) MemGuard-BR+PS

Figure 3.15: Frame-rate comparison. The weight assignment is 1:2:4:8 (Core0,1,2,3) and $r_{min} = 1.2$GB/s for MemGuard-BR+SS and MemGuard-BR+PS.

is 1:2:4:8 (for Core0,1,2,3 respectively) and the r_{min} is 1.2GB/s. Vanilla is unaware of the weight assignment. Hence, all instances show almost identical frame-rates. MemGuard-BR+SS enforces bandwidths up-to the guaranteed bandwidth r_{min}, hence it shows different frame-rates. However, due to the nature of *fps*, mostly sequential memory accesses which can achieve peak bandwidth, the reserved bandwidth is relatively small, compared to the peak-bandwidth. Therefore, the frame-rate ratio is not similar to the weight assignment. On the other hand, MemGuard-BR+PS shows frame-rates that is almost identical to the assigned weight. This is because MemGuard-BR+PS enforce the given bandwidth assignment all the time (by starting a new period immediately after cores use the r_{min}), resulting better prioritization over MemGuard-BR+SS.

3.7 Related Work

Resource reservation has been well studied especially in the context of CPU scheduling [88, 37] and has been applied to other resources such as GPU [93, 94]. The basic idea is that each task or a group of tasks reserves a fraction of the processor's available bandwidth in order to provide temporal isolation. Abeni and Buttazzo proposed Constant Bandwidth Server (CBS) [37] that implements reservation by scheduling deadlines under EDF scheduler. Based on CBS, many researchers proposed reclaiming policies in order to improve average case performance of reservation schedulers [89, 90, 91, 92]. These reclaiming approaches are based on the knowledge of task information (such as period) and the exact amount of extra budget. While our work is inspired by these works, we apply reclaiming on memory bandwidth which is very different from CPU bandwidth in many ways.

DRAM bandwidth is different from CPU bandwidth in the sense that achieved bandwidth depends on the DRAM state and access history which makes it difficult to guarantee performance. To solve this problem, several DRAM controllers were proposed. Akesson et al. proposed Predator DRAM controller that uses combination of regulators and credit based scheduler to provide performance guarantee among multiple hardware components that access the DRAM [51, 47]. Reineke et al. proposed PRET DRAM controller that partitions the physical address space based on the internal structure

of the DRAM chip in order to eliminate contention caused by sharing such internal resource [53]. Also in more general purpose computing systems, DRAM controller is studied in order to improve fairness and throughput. Nesbit et al. applied the network fair queuing theory in designing DRAM controller [32]; Ebrahimi et al. proposed a cache controller level throttling mechanism [67], which is similar to our method in the sense that it effectively changes request rates. While these DRAM controllers provide solutions to improve predictability and isolation in hardware level, we focus on a software level solution that can be applied to commodity hardware platforms.

OS level memory access control was first discussed in literature by Bellosa [65, 66]. Similar to our work, this work also proposed a software mechanism—increasing/decreasing the number of idle loops in the TLB miss handler—in order to control memory contention. There are, however, three major limitations, which are addressed in our work: First, it defines the maximum reservable bandwidth in an ad-hoc manner—i.e., $0.9 \times StreamB/W$—that can be violated depending on memory access patterns as shown in Section 3.1; Second, it does not address the problem of wasted memory bandwidth in case cores do not use their reserved bandwidth. Finally, it is designed for soft real-time applications guarantees as it allows cores to overuse their reserved bandwidth until a control mechanism stabilize the cores' bandwidth usages by increasing/decreasing the idle loops in the TLB handler. It is, therefore, not appropriate to apply the technique for hard real-time applications. In contrast, our work in Chapter 3 clearly defines the maximum reservable bandwidth based on understanding of DRAM and the DRAM controller, provides stronger fine-grained bandwidth guarantee for hard real-time applications, and provides reclaiming and sharing mechanisms to better utilize the memory bandwidth while still providing a minimum bandwidth guarantee to each core.

3.8 Summary

In this chapter, we have presented MemGuard, a memory bandwidth reservation system, for supporting efficient memory performance isolation on multicore platforms. It decomposes memory bandwidth as two parts, guaranteed bandwidth and best effort bandwidth. Memory bandwidth reservation is

provided for the guaranteed part for achieving performance isolation. An efficient reclaiming mechanism is proposed for effectively utilizing the guaranteed bandwidth. It further improves system throughput by exploiting best effort bandwidth after each core satisfies its guaranteed bandwidth. It has been implemented in Linux kernel and evaluated on a real multicore hardware platform.

Our evaluation with SPEC2006 benchmarks showed that MemGuard is able to provide memory performance isolation under heavy memory intensive workloads. It also showed that the proposed reclaiming and sharing algorithms improve overall throughput compared to a reservation only system under time-varying memory workloads.

CHAPTER 4

RESPONSE-TIME ANALYSIS FOR MEMORY BANDWIDTH REGULATED SYSTEMS

We now turn our attention to the response-time analysis in memory bandwidth regulated systems.

Data intensive workloads, which require frequent memory accesses, are increasingly more pervasive in modern embedded computing systems including critical real-time systems. For example, an aircraft now processes massive vision data in real-time to track objects in flight [101]. Processing such massive data requires more computing power. Therefore, there is a growing need for powerful multiprocessor to consolidate such workloads.

Consolidating data intensive tasks together with critical real-time tasks, however, poses a significant challenge due to interference on shared resources such as system bus and memory. It becomes more apparent as core count and memory intensity of tasks increase. The authors of [5] shows that a task can suffer 300% WCET increase due to memory interference even when tasks spend only 10% of their time on fetching memory in an eight core system.

As reviewed in Chapter 2, one solution is to use specialized hardware which has capabilities to control such interferences. For example, a predictable DRAM controller [47] can provide guaranteed bandwidth and latency on accessing memory. Similarly a TDMA based system bus [102] can provide timing guarantee on accessing the shared bus. Hardware based approaches, however, prevent us from using cost effective commercial off-the shelf (COTS) components.

We propose to use a software based memory bandwidth throttling mechanism to explicitly control the memory interference. The basic idea of memory throttling is periodically limit the amount of memory accesses similar to aperiodic servers for CPU bandwidth reservation [36]. The throttling implementation we use in this chapter is similar to the MemGuard system, described in the previous chapter, but differs in that the regulation periods of the cores are not synchronized and it does not supports reclaiming and sharing.

Using this mechanism, we are interested in protecting critical tasks from non-critical tasks where tasks are partitioned based on their criticality. As a first step, we consider a scenario where critical tasks run on a single core, we call *critical core*, and non-critical tasks run on the rest of the cores, we call *interfering cores*, as shown Figure 4.1. The interfering cores are, however, throttled using our memory throttling mechanism. Our goal is to find the throttling parameters, namely budgets for a given period value, on the interfering cores that satisfies schedulability of tasks on the critical core while minimizing performance impact of tasks on the interfering cores.

On throttling multiple interfering cores, we consider a static and a dynamic throttling strategies which differs in how budget is allocated in each period. We describe algorithms to get analytic solution on computing throttling parameters. We implemented the throttling mechanism on Linux kernel by extending standard group scheduling interface. We experimentally validate how the computed throttling parameters affect execution time of tasks on the critical core. We also compare the effect of static or dynamic throttling strategies in terms of slowdown experienced due to throttling on the interfering cores.

Although using throttling neither increases memory bandwidth of the hardware, nor reduces the memory access requests from tasks, it provides isolation for critical core among multiple cores. Our software based approach allows us to use COTS components and does not require any modification on the existing applications.

This chapter is organized as follows: Section 4.1 formally defines system model and the problem. Section 4.2 describes solution in the case of single interfering core. Section 4.3 extends the results to multiple interfering cores. Section 4.4 shows experiment results.

Section 4.5 discusses the limitation and future work. We review related work in Section 4.6 and conclude in Section 4.7.

4.1 System Model

We consider a multiprocessor architecture as shown in Fig. 4.1 where system bus and memory are shared among cores and each core has its private cache. We assume that cache miss is synchronous in the sense that whenever there

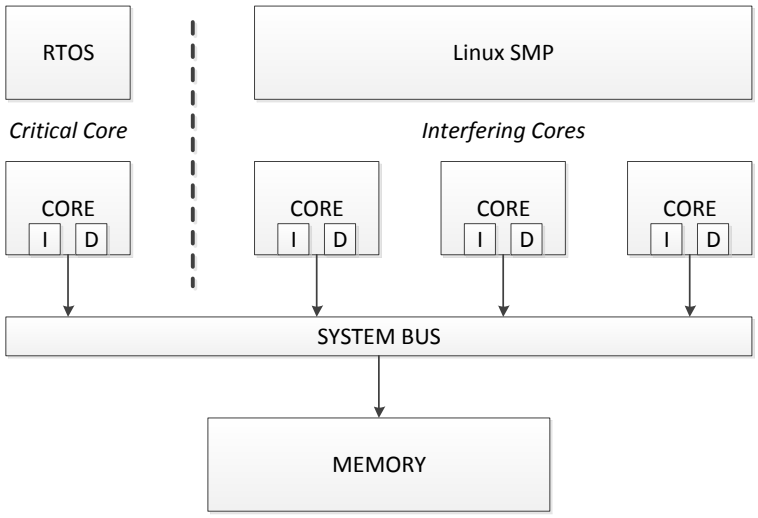


Figure 4.1: System model.

is a miss, the core is stalling until the cacheline is fetched from the memory. There is only one DRAM controller connected to the system bus, which is a common system configuration for embedded systems. For the purpose of analysis, we assume each memory access latency is a constant when there are no contending concurrent accesses. We further assume the arbitration scheme in the DRAM controller is round-robin. Hence, the service rate of a core is deterministic as a function of contenting requests from other cores. These assumptions regarding DRAM controller can be realized with using a predictable DRAM controller such as [48].

We categorize cores into two groups, a core under analysis which we call a *critical core* and *interfering cores*. For the core under analysis, we assume that a fixed priority preemptive scheduler is used to schedule tasks. We assume WCET and the worst case number of cache misses for each task is given a priori. These parameters can be obtained from static analysis or from measurement by running in isolation. We assume preemption does not affect the number of cache-misses of a task, for example by partitioning cache to each task.

On the core under analysis, i.e., *critical core*, a set $\mathcal{T} = \tau_1, \ldots, \tau_n$ of n periodic real-time tasks are scheduled with a deadline monotonic scheduling algorithm; the period and the relative deadline of τ_i is denoted by T_i and D_i ($D_i < T_i$). The WCET of a task is denoted by C_i and the number of worst

case cache misses is given by CM_i. Note that the CM_i does not necessarily coincide the worst case execution path. We denote the subset of tasks with priority higher/lower than task τ_i with $hp(i)/lp(i)$, and furthermore, let $hep(i) = hp(i) \cup \tau_i$.

On the *interfering cores*, the collective memory access time of the cores is regulated with the period P and the budget Q. Note that Q denotes time to access memory. As we assume constant time for accessing DRAM and each access is restricted to be a cache-line granularity, Q can be converted to the number of memory accesses (i.e., cache misses) and vice versa.

The budget is distributed among the interfering cores either (1) statically at the beginning of each period, or (2) dynamically at runtime based on the demand of each core. We made no assumption about the scheduling policies on the interfering cores. In other words, any scheduling algorithm can be used (e.g., CFS in Linux).

The goal is to find the throttling configuration, budget Q values for a given period P, for throttled cores such that satisfies schedulability of critical tasks on the critical core while minimizing slowdown of tasks running on the interfering cores.

4.2 Single Interfering Core

In this section we consider a simple case where there are only two cores: one is the core under analysis with critical tasks assigned to it and the other is the interfering core. We later extend our analysis to consider multiple interfering cores in the next section.

We first describe how memory contention from the interfering core can be estimated based on the throttling parameters as presented in [103]. We then extend the response time analysis by taking into account the task stalling caused by contention from interfering cores. Finally, we formulate a problem of finding a throttling budget Q for a given period P of the interfering core such that the schedulability of the tasks assigned on the critical core is satisfied.

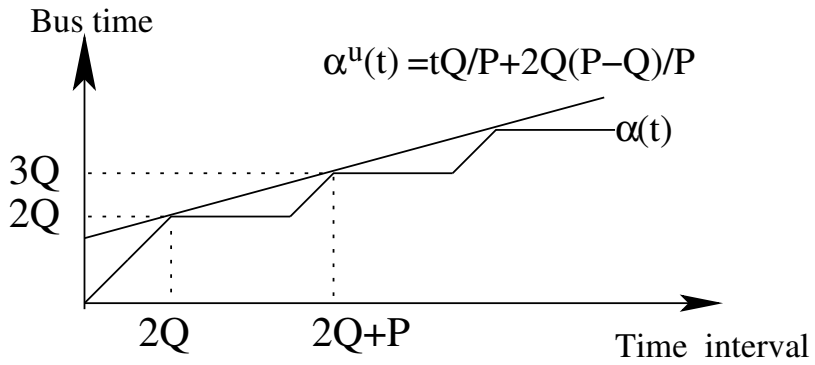


Figure 4.2: Arrival curve and its upper bound for one throttled core with P as period and Q as the cache-miss budget.

4.2.1 Flow "reshaping" by throttling

In order to account the task stalling due to the contention on the shared memory, we need to know the memory access pattern of the task. However, this turns to be rather complicated since it heavily depends on the dynamic task execution as well as the scheduling algorithm applied on the system. Prior work either assumes a specific memory access pattern [104] or a static cyclic scheduler [5, 102], however, these approaches suffer limitations when applying to the real applications.

The throttling mechanism provides another alternative to account the memory accesses from the interfering core. The throttling controller works independently of the specific scheduler and task set on the throttled core, it simply stalls the task on the throttled core when the budget is consumed within the period.

The arrival curve of the flow can be derived similar to the method as commonly found in [98], i.e., the maximum possible traffic amount $\alpha(t)$ for a given time interval t. In the worst case the core can generate up to $2Q$ continuous cache misses in a time window with $2Q$ length due to the back logged one Q from the previous period, and then another Q every period. The derived flow arrival curve is shown in Figure 4.2. Notice that this curve is a step function and the upper bound of the arrival curve, $\alpha^u(t)$, is also depicted in the figure.

4.2.2 Stall time calculation

As we assumed that the inter-connection network to memory is bus and its arbitration is based on round robin, each memory access could be delayed by one memory access from the other core. The stall that one task can suffer depends on both the number of memory accesses this task needs to perform, as well as the number of memory accesses generated on the other core during this task's execution. To this end, we show how these two factors and the task stall can be accounted.

Let us consider a task which generates CM memory accesses in worst case. Since each access can be delayed by interfering core's access, the maximum stall time this task can suffer is upper bounded by $CM \cdot L$ where L is the time needed to perform one memory access. It represents the task interior requirement on the memory resource.

The task stall due to the memory accesses from the other core can be estimated by accounting the memory access traffic during this task's execution. However, unlike the CM value which does not change depending on its stall time, there is a circular dependency between the task stall time and the number of memory accesses from the other core. Figure 4.3[1] illustrates the stall for one task with worst case execution time C, measured in isolation, and a throttled core with arrival curve $\alpha(t)$. The amount of traffic $d^1 = \alpha(C)$ during the task's execution could interfere this task and cause an increase of d^1 in its execution time. However, the increased execution time, $C + d^1$, would possibly suffer a higher level of memory traffic interference, which is equal to $d^2 = \alpha(C + d^1)$. This process continues until it converges. As clearly showed in the figure, this can be formulated as an iterative procedure, it terminates and returns the stall for this task when the procedure is converged at Δ, that is

$$d^{(k+1)} = \alpha(C + d^{(k)}) \tag{4.1}$$

Finally, the task stall cannot exceed one of these two factors—cache misses of the task under analysis and the interfering flow of the other core, hence, the stalled execution time $\widehat{C}$ for one task can be expressed as the solution of

[1]The time length in the figure is selected to better demonstrate the iterative procedure of task stall calculation. i.e., $\Delta \leq C$ and $\alpha(t) \leq t$ in Figure 4.3.

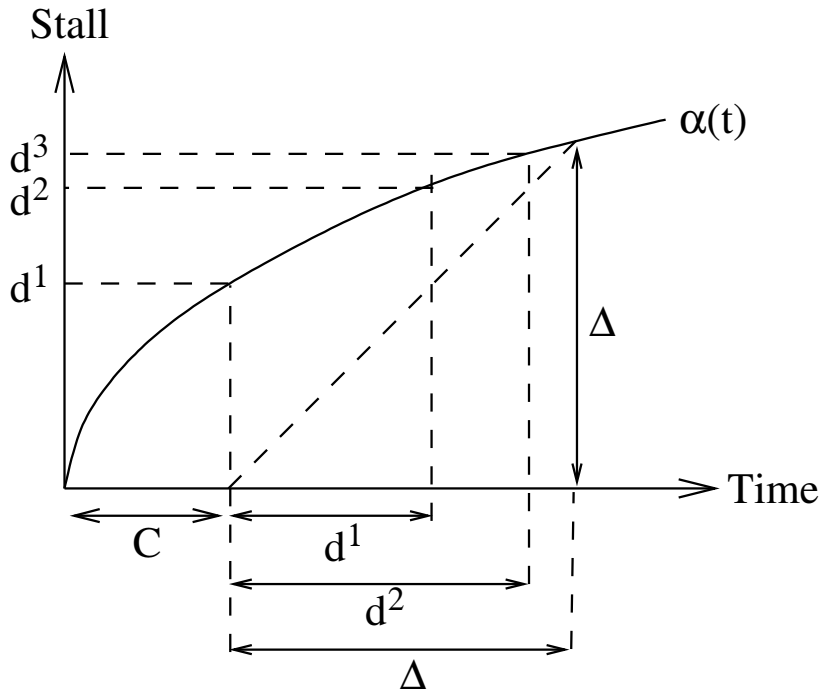


Figure 4.3: The circular dependency between the traffic from throttled core andthe stall of task on the critical core, d represents the amount of traffic for a given time interval while Δ represents the overall stall for this task.

this iterative procedure:

$$\widehat{C^{(k+1)}} = C + \min\{CM \cdot L, \alpha(\widehat{C^{(k)}})\} \tag{4.2}$$

4.2.3 Extended response time analysis

With the stalled execution time for each task, we can perform the classical response time analysis [105]. However, this turns out to be quite pessimistic as each task under analysis is assumed to possibly suffer the $2Q$ delay. In reality, this cannot happen since for a continuous time interval, there could be only one back-logged Q at the beginning. We still use the iterative response time analysis; refine it by adding another term to account the delay due to the memory contention with other core. The main intuition is that, instead of applying the delay upon each single task, we directly compute the delay for the response time at each iteration.

The iterative response time calculation is expressed as follows:

$$R_i^{(k+1)}=C_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{R_i^{(k)}}{T_j}\right\rceil\cdot C_j+\min\{\mathcal{N}(R^{(k)})\cdot L,\alpha(R^{(k)})\} \tag{4.3}$$

where

$$\mathcal{N}(t)=CM_i+\sum_{\tau_j\in hp(i)}\left\lceil\frac{t}{T_j}\right\rceil\cdot CM_j. \tag{4.4}$$

When ignoring the third term on the RHS of the equation, this is exactly the same as the classical response time analysis. The new introduced third term represents the maximum possible stall upon all tasks executing (include τ_i itself and the preempting tasks) during the time interval R^k. Specifically, number $\mathcal{N}(R^k)$ captures the total number of cache misses among all tasks executing within this time interval R^k, whereas $\alpha(R^k)$ is the total memory access traffic from the throttled core during R^k. Following the stall analysis for one task in previous subsection, we know that the third term (the min function) is the total stall caused by memory contention between all the executing tasks and the traffic from the throttled core within this R_i time interval. The response time of τ_i is obtained when R^k converges as the classical response time analysis.

4.2.4 Calculation of throttling budget

Given a throttling period P on the throttled core, we now consider how to calculate the maximum budget value Q such that the tasks assigned on the critical core can have their deadlines guaranteed. We first assume that tasks are schedulable when they run in isolation. We also assume the tasks on the critical core are scheduled according to fixed priority assignment with τ_1 being the highest priority task, the budget Q for the throttled core can be calculated by considering the feasibility of individual task on the critical core, in decreasing priority order. Let Q_i denote the maximum budget value such that the subset $hep(i)$ on critical core have their deadline satisfied. The following properties can be derived.

Property 1. The Q_i sequence is monotonically non increasing.

Property 2. The maximum budget that can be assigned to the throttled core is Q_n where n is the number of tasks on the critical core.

Proof. Property 1 can be easily proved by contradiction. Suppose for a task τ_i, there exists one task $\tau_j \in lp(i)$ with Q_j value larger than Q_i. Since Q_j is the value such that the task set $hep(j)$ is feasible, hence task subset $hep(i) \in hp(j)$ is also feasible. This contradicts the assumption that Q_i is largest value such that $hep(i)$ is feasible.

Property 2 follows directly from Property 1. □

The overall algorithm to compute the budget Q is presented in Algorithm 1. Line 2 initializes the Q value to the period P; this serves as the upper bound of Q. Then the tasks on the critical are checked in decreasing priority order: if the task is verified to be feasible by computing its response time, then the algorithm moves on to the next task; otherwise, the value Q_i is calculated and Q is updated with Q_i. Notice that since the algorithm follows the priority order, $hp(i)$ is guaranteed to be feasible when checking task τ_i, hence Line 5 only needs to deal with task τ_i alone. Finally, when the algorithm finishes all the tasks, it returns the value Q as the maximum budget for the throttled core.

Input: The throttling period P and the taskset parameters on the critical core.
Output: The maximum budget value Q such that the critical core is feasible

```
1 begin
2     Q = P ;
3     for i ← 1 to n do
4         Calculate the response time R_i according to Equation (4.3) ;
5         if R_i > D_i then
6             Calculate Q_i such that τ_i is feasible ;
7             update Q with Q = Q_i;
8     return Q ;
```

Algorithm 1: Calculate budget Q such that the critical core is feasible.

Now it remains to explain how to calculate Q_i to make τ_i feasible, as the Line 5 in the algorithm. Notice the response time calculation depends on both the arrival times of high priority tasks and the delay caused by the throttled core during the response time interval, which makes the Q calculation not straightforward. Since for a given traffic delay function $\alpha(t)$, the response time function for τ_i has dis-continuous points only at the arrival time of high

priority task. We denote this set of time instants as the testing set as in the following equation.

$$\mathcal{TS}(\tau_i) \doteq \{t | t \in [C_i, D_i] \cap t = k \cdot T_j \ \forall \tau_j \in hp(i),\ k \in \mathbb{N}\} \tag{4.5}$$

For a certain time point from $\mathcal{TS}(\tau_i)$, the number of preemptions on τ_i is a fixed value independently of the traffic function, which allows us to solve the $\alpha(t)$ function in terms of budget Q. The Q can be calculated by the following equation:

$$Q_i(t) = \begin{cases} \emptyset; & \text{if } \mathcal{S}(t) \leq 0 \\ \infty; & \text{if } \mathcal{S}(t) \geq \mathcal{N}(t) \cdot L \\ \frac{2P+t}{4} - \frac{((2P+t)^2 - 8\mathcal{S}(t)P)^{1/2}}{4} & \\ & \text{if } 0 < \mathcal{S}(t) < \mathcal{N}(t) \cdot L \end{cases} \tag{4.6}$$

where $\mathcal{S}(t)$ is the maximum allowed stall time for τ_i at time t that still satisfies the schedulability constraints and can be expressed

$$\mathcal{S}(t) \doteq t - C_i - \sum_{\tau_j \in hp(i)} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j. \tag{4.7}$$

Proof. The first case is when the task execution plus the interference due to the preemptions from high priority tasks already exceeds the time interval t, hence, there is no solution at this point. On the other hand, the second line shows the situation when the cache misses from the tasks executing on this core is small or the task slack is big enough, such that the delay bound from the task itself is enough, regardless of the traffic flow on the other core.

The throttling mechanism plays an important role in the third case: we have to limit the memory access traffic from the other core by controlling the budget Q so that the delay on this task would not cause deadline miss. With the upper bound of traffic function as shown in Fig. 4.2, we have

$$\alpha(t) \leq \alpha^u(t) = t\frac{Q}{P} + \frac{2Q(P-Q)}{P}.$$

hence it is enough to guarantee that

$$t\frac{Q}{P} + \frac{2Q(P-Q)}{P} \leq \mathcal{S}(t). \tag{4.8}$$

Solve this equation and discard the unfeasible solution we get

$$Q \leq \frac{2P+t}{4} - \frac{((2P+t)^2 - 8\mathcal{S}(t)P)^{1/2}}{4}.$$

This proves the Equation (4.6). □

Notice that τ_i is feasible if there exists one point from $\mathcal{TS}(\tau_i)$ that satisfies Equation (4.3), therefore, the budget value Q_i that guarantees the feasibility of τ_i can be expressed as:

$$Q_i = \max_{t \in \mathcal{TS}(\tau_i)} Q_i(t) \tag{4.9}$$

where $Q_i(t)$ is computed according to Equation (4.6).

With this calculated Q_i for each task, now we can follow Algorithm 1 to calculate the budget Q for the throttled core with the schedulability of the tasks on the critical core guaranteed.

4.3 Multiple Interfering Cores

Having shown the interference from one single core to the critical core, now we extend the result to the case when the system contains several throttled cores, which both contend for the access to the main memory with the critical core. This is the situation shown in Figure 4.1. Depends on how the budget is distributed among all the throttled cores, we propose two different throttling schemes: *static budget distribution*, which assigns fixed amount of budget to each core, and *dynamic budget distribution*, which assigns dynamically assign budget on-demand basis. For each scheme, our goal is to find a budget assignment for throttled cores that maximizes utilization of throttled cores that satisfy schedulability of the critical core.

4.3.1 Static budget distribution

In this scheme, we consider each core owns its budget which is statically distributed from a global budget and all cores share the same period. We assume the static distribution proportion for each throttled core is given in

priori by, for example, analyzing the characteristics of tasks on each throttled core.

First, we describe method to compute the stall of one task on the critical core. Assuming each throttled core has an individual budget, it is easy to see the upper bound of its memory access can be computed by the corresponding period and budget. Similar to the analysis presented in the previous section, we denote the arrival curve for each throttled core by $\alpha_c(t)$, where $c = 1, \ldots, M$ and M is the number of throttled cores. The increased execution time of one single task τ_l on the critical core, denoted by $\widehat{C}_l$, can be computed by the following iterative way:

$$\widehat{C_l^{(k+1)}} = C_l + \sum_{1 \le c \le M} \min\{CM_l \cdot L,\ \alpha_c(\widehat{C_l^{(k)}})\} \tag{4.10}$$

The stalled execution time is calculated by summing up all the delays caused by each throttled core and its original execution time. The delay factor from each core is determined similar to Equation (4.2) as analyzed in the section before: the cache misses due to the task itself or the memory traffic flow from the throttled cores. When the iterative procedure converges, it returns the increased execution time, which includes the original execution requirement plus maximum delay from all throttled cores during this increased execution time.

The response time analysis can be extended to the multi throttled core case with the similar technique used for the single throttled core. The main intuition is that, at each iteration we should consider delay from each throttled core and sum them up, as showed in the next equation.

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + \sum_{1 \le c \le M} \min\{\mathcal{N}(R^{(k)})L, \alpha_c(R^{(k)})\} \tag{4.11}$$

, where $\mathcal{N}(t)$ is defined in Equation (4.4) and c is the index of throttled core.

We assume throttled core with index $c \in [1, M]$ is assigned a fixed ratio r_c of the global budget, i.e., the budget for this core is $r_c Q_i$. Furthermore, let the throttled cores are indexed by increasing ratio order. Given the distribution proportion of global budget, how to calculate the budget still turns out to be a problem not easy to solve. The tricky part is that when summing up

the delay factor from each throttled core, it is obtained either from the task cache miss (as the first item in the min function) or the throttled core flow (as the second item in the min function), depending on the specific arrival curve of this throttled core and the time instant.

To consider the schedulability of one task $\tau_i, i \in [1, N]$, on the critical core, we get the testing set $\mathcal{TS}(\tau_i)$ for τ_i as in Equation (4.5) and the slack $\mathcal{S}(t)$, as defined in Equation (4.7), at one specific time instant in the testing set. When considering the schedulability of the task τ_i, for a given time instant $t \in \mathcal{TS}(\tau_i)$ and one throttled core with index $c \in [1, M]$, we calculate the ranges of global budget so that we can determine the delay from this throttled core is obtained from which factor. Notice that because the throttled cores are indexed by the order of budget ratio, therefore the arrival function, we can now determine the delay factor for the remaining throttled cores. We solve the equation of the $Q_i^{c,t}$ and consider all the indexes of c, t, to get the value of Q_i to make task τ_i on the critical core schedulable. Finally, we merge all the solutions for each task τ_i on the critical core to get the final result of Q.

The computation process is explained in detail now. We need to distinguish three cases depends on the range of value $\mathcal{S}(t)$:

When $\mathcal{S}(t)$ is no larger than 0. In this case, the task on the critical core could not suffer any delay *at this time point*. Therefore, the budget would be zero.

When $\mathcal{S}(t)$ is no less than $M \cdot \mathcal{N}(t)L$. In this case, no matter how large the traffic flow would be, since the delay from each core is bounded by $\mathcal{N}(t)L$, the budget can be assigned arbitrarily.

We focus on the case when $\mathcal{S}(t)$ is within the range between the two previous values. The main idea is to divide the throttled cores by the return value of the min function in Equation (4.11). Since the throttled cores are indexed by increasing budget ratio, there may exist one core with index c such that, for a specific time instant t and a global budget Q_i, the delay from the throttled core with a index no less than c is obtained from the task cache miss term. These cores have even higher traffic flow and the min function would return the task cache miss item. Now consider the min function and replace $R^{(k)}$ by t we have:

$$\alpha_c(t) \geq \mathcal{N}(t)L$$

where $\mathcal{N}(t)$ is defined in Equation (4.4). Notice the $\alpha_c(t)$ function is a step function and its upper bound $\alpha_c^u(t)$ is used to simplify the computation.

$$\alpha_c^u(t) = t\frac{r_c Q_i}{P} + \frac{2r_c Q_i(P - r_c Q_i)}{P} \geq \mathcal{N}(t)L$$

Solving this equation, let $Q_i^L(c,t)$ denote the lower bound of Q value such that stall caused by core c at time t is obtained from the cache miss part, we have:

$$Q_i^{c,t} \geq Q_i^L(c,t) = \frac{r_c(2P+t)}{4r_c^2} - \frac{(r_c^2(2P+t)^2 - 8r_c^2 L\mathcal{N}(t)P)^{1/2}}{4r_c^2}. \tag{4.12}$$

Now that we know if $Q_i^{c,t}$ is larger than $Q_i^L(c,t)$, the core with index no less than c would cause delay equal to the cache miss part, which is the key idea to solve the problem. we group the cores into the two sets depending on their flow value. Specifically, the first $c-1$ throttled core contribute stall from the traffic flow part, while the remaining cores (starting from index c inclusive) contribute $\mathcal{N}(t)L$ each. Now we are ready to tackle Equation (4.11). The third item of the RHS with $R^{(k)}$ replaced by t is:

$$\sum_{1 \leq c \leq M} \min\{\mathcal{N}(t)L, \alpha_c(t)\} = \sum_{1 \leq j < c} \alpha_j(t) + (M - c + 1)\mathcal{N}(t)L$$

Therefore, Equation (4.11) can be rewritten as

$$\sum_{1 \leq j < c} \alpha_j(t) + (M - c + 1)\mathcal{N}(t)L \leq \mathcal{S}(t)$$

where $\mathcal{S}(t)$ is defined as in Equation (4.7)

Now the solution becomes similar to the single throttled core case: we expand the summation part α_j for each throttled core with r_j and Q_i and sum up. Let $\widehat{\mathcal{S}(t)} = \mathcal{S}(t) - (M - c + 1) * \mathcal{N}(t)L$, $\widehat{r_c} = \sum_{1 \leq j < c} r_j$ and $\widehat{r_c^*} = \sum_{1 \leq j < c} r_j^2$. Then we are ready to solve the following equation to get Q:

$$t\frac{\widehat{r_c} Q_i}{P} + \frac{2\widehat{r_c} Q_i P - 2\widehat{r_c^*} Q_i^2}{P} \leq \widehat{\mathcal{S}(t)}.$$

Let $Q_i^H(c,t)$ denote the upper bound of Q value this equation is satisfied.

We get:

$$Q_i^{c,t} \leq Q^H(c,t) = \frac{\widehat{r_c}(2P+t)}{4\widehat{r_c^*}} - \frac{(\widehat{r_c}^2(2P+t)^2 - 8\widehat{r_c^*}\mathcal{S}(t)P)^{1/2}}{4\widehat{r_c^*}} \tag{4.13}$$

Put Equation (4.12) and Equation (4.13) together and consider all indexes of throttled cores and all time points in testing set. Since task τ_i is schedulable as long as there is one throttled core c and one time instant $t \in \mathcal{TS}(\tau_i)$ satisfied, we have:

$$Q_i = \bigcup_{1 \leq c \leq M} \bigcup_{t \in \mathcal{TS}(\tau_i)} \{Q_i^{c,t} \geq Q_i^L(c,t) \bigcap Q_i^{c,t} \leq Q_i^H(c,t)\} \tag{4.14}$$

where $Q_i^L(c,t)$ and $Q_i^H(c,t)$ are solved in Equation (4.12) and Equation (4.13) respectively.

Finally, considering the multiple tasks on the critical core, the final result on Q is the intersection of all Q_i ranges.

$$Q = \bigcap_{1 \leq i \leq N} Q_i$$

where each Q_i is calculated in Equation 4.14.

4.3.2 Dynamic budget distribution

In this scheme, all throttled cores share a single global budget and a period. When each core accesses memory, it consumes the global budget. When the global budget is exhausted, tasks on all throttled cores are suspended until the next period begins.

The dynamic budget distribution scheme reduces the possibility of throttling on the throttled cores because budget is consumed more efficiently on-demand basis. Therefore it improves responsiveness of applications on the throttled cores. It becomes more evident when tasks have high variance on memory access requests. On the other hand, however, it makes analysis more difficult, because budget distribution among cores keeps changing over time.

A safe, but pessimistic, upper bound of delay function is

$$\widehat{C^{(k+1)}} = C + \min\{M \cdot CM \cdot L, \alpha(\widehat{C^{(k)}})\} \tag{4.15}$$

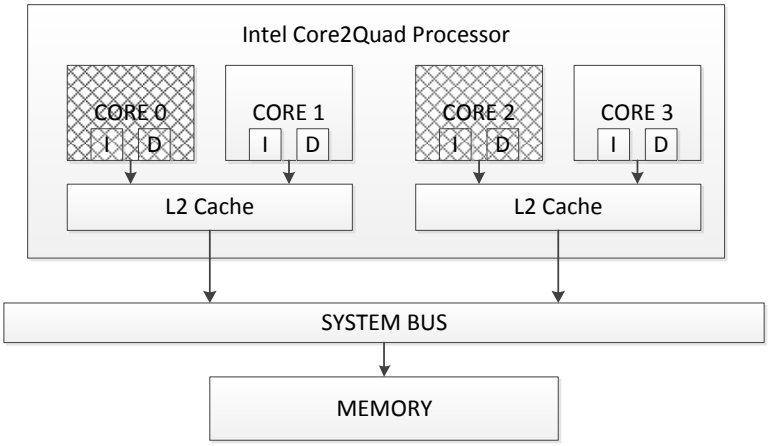


Figure 4.4: Architecture of our evaluation platform.

where M is the number of interfering cores. This is identical to Equation (4.2) except that M is multiplied in the first part of the min function. This is because the critical task on the critical core now can wait up to M arbitration.

Similarly, we can use Equation (4.3) for response time analysis just by multiplying M to the first part, $\mathcal{N}(R^k)$, of the min function.

Finally, computing the global budget Q can be obtained by using Equation (4.6), again multiplying M to all $\mathcal{N}(t)L$.

Note that this analysis is more pessimistic than the analysis in Section 4.3.1, because we do not consider each individual flow, which could possibly produce a tighter bound.

4.4 Evaluation

In this section, we experimentally evaluate our approach in terms of isolation guarantee of the critical core and performance impact of throttled cores (i.e., interfering cores).

4.4.1 Software Implementation

The software implementation we used in this chapter is different from the MemGuard system described in Chapter 3 in several ways [2]. It uses Linux group scheduling interface called *cgroup* instead of directly manipulating per-core runqueue as in MemGuard. The cgroup interface is originally designed to specify fraction of system resources such as CPU cores and memory capacity to a group of tasks that may span multiple cores [106]. We extended the cgroup interface so that we can specify memory bandwidth with a pair of period and budget. This interface gives us the maximum flexibility. For example, we can enforce memory throttling (1) for each individual core by creating a cgroup for each core (Section 4.2), or (2) for a group of cores by creating a single cgroup and assigning the cores to the cgroup (Section 4.3).

On the other hand, precise per-core control is more difficult because of the complexity of group scheduling implementation in Linux as well as hardware restrictions—For example, it is not easy, if not impossible, to stop multiple cores at once when they collectively used the budget, which is shared by all cores in the cgroup. Hence, our implementation used in this chapter periodically polls the hardware performance counters, instead of generating interrupts, to account the budget consumption. The polling occurs at every 1ms or context switching time, whichever comes first. Therefore, there can be some amount of jitter from the time when the budget is expired to the time when scheduler actually performs the throttling operations.

4.4.2 Evaluation Setup

The testbed contains Intel Core2Quad Q8400 processor running at 2.66GHz shown in Figure 4.4. Core0-1 share a 2MB L2 cache, and Core2-3 share another 2MB L2 cache. L2 caches are directly connected to the shared FSB running at 1333MHz. For this evaluation, we only use Core1 and Core3 for the experiment in order to eliminate the effect by sharing L2 cache [3]. We use last level cache (LLC) miss count, reported from the hardware performance counter of each core, as the cache-miss number. We obtained the memory

[2]The source code can be found in https://github.com/heechul/linux-sched-coreidle/tree/sched-3.2-throttle-v2

[3]We disabled cpu cores because the processor we used does not support disabling L2 cache.

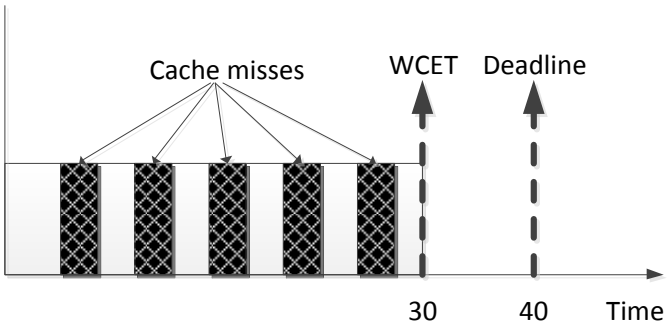


Figure 4.5: Task under analysis on the critical core.

access latency L, using a synthetic benchmark [4]. We convert buget Q to the number of cache misses with Q/L.

4.4.3 Response Time on the Critical Core

We present measured response time on the critical core while varying memory bandwidth on the interfering core. In this experiment, we use Core1 as a *critical core* and Core3 as an *interfering core*. The interfering core is regulated by the throttling mechanism with period P, and budget Q.

Fig. 4.5 shows the task under analysis,τ_{crit}, that runs on Core1. It is engineered to take 30ms to finish when run in isolation; it spends 50% of time stalling on cache-misses. The cache-misses are placed in 10 equally spaced chunks; each chunk generates 1.5ms of continuous cache-misses. Note that we engineered each memory access to cause a cache-miss during the chunk. Then the task spends another 1.5ms for pure computation without any cache-miss.

On Core3 another engineered task, τ_{inter}, is running. It generates continuous cache-misses but throttled with P and Q values. We measured response time of τ_{crit} on the critical core, while varying the throttling memory bandwidth of the interfering core.

Figure 4.6 shows the response time impact of throttling. X-axis shows the allocated memory bandwidth on the interfering core, Core3; the throttling period P is set to 10ms and budget, Q, is varied so that the bandwidth

[4]It performs read and write operation on a 8MB array for every 8 cache-line distance

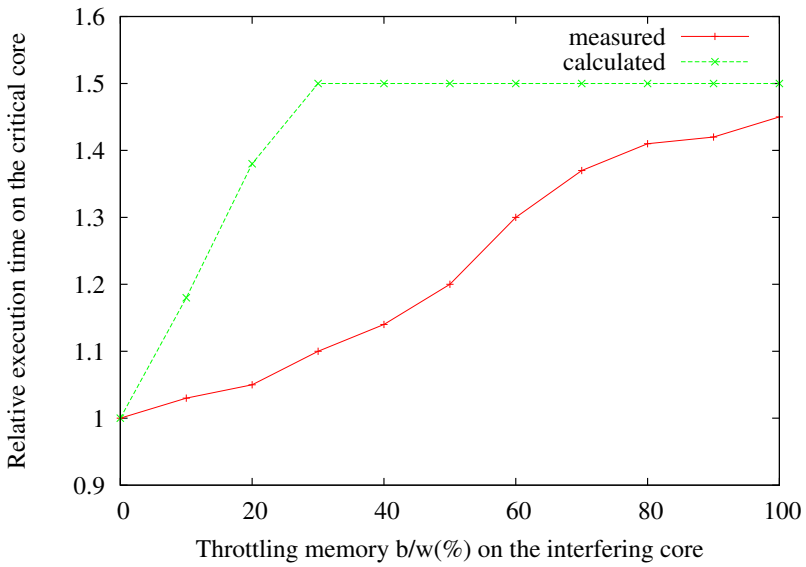


Figure 4.6: Impact of throttling bandwidth to the response time on the critical core.

changes from 0 to 100%. Note that zero bandwidth is equivalent to the case when nothing is running on the interfering core. Y-axis shows the relative response time increase of T_{crit} on the critical core. The *calculated* curve shows the calculated response time of T_{crit} according to Eq. 4.2. The *measured* curve shows the measured worst case response time among 1000 repeated invocations of T_{crit}; We randomly vary the interval time between two successive invocations from 0 to 30ms in order not to be affected by regularity of interfering flow on the interfering core.

In calculated curve, response time increases as the assigned bandwidth of the interfering core increases. When the assigned bandwidth increases above 30%, resulting response time saturates because from that point it is bounded by the number of cache-misses of the T_{crit} itself. In measured curve, response time also increases as bandwidth increases, but it is slower than the calculated curve. Notice that response time increase is sharper as assigned bandwidth is above 50% and approaches to the calculated bound. The difference between the calculated and measured curve show pessimism of our analysis. In the analysis, we considered the worst case scenario where every cache-miss from t_{crit} is delayed from cache-misses of the interfering core. However, the probability of such worst case scenario is low in a real situation as suggested by the measurement result.

Scheme	Total throttled time
Static	9,689 ms
Dynamic	4,622 ms

Table 4.1: Throttling time of interfering cores.

4.4.4 Performance Impact on Throttled Cores

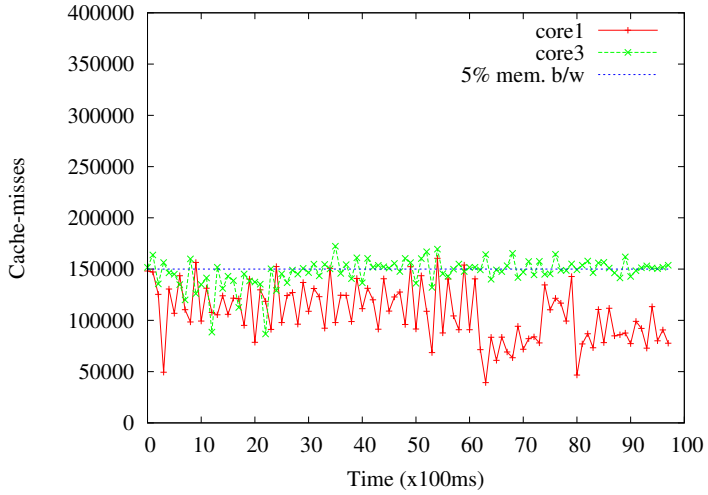
In this experiment, we compare performance impact of throttled cores under static and dynamic budget distribution schemes with realistic workload.

We use both Core1 and Core3 as interfering cores. In this experiment, we do not have a critical core, since we focus on the impact of budget distribution schemes on the throttled cores. We use two mpeg4 video streams, 720p and 1080p movie trailers, as workloads and played them on Core1 and Core3 respectively using *mplayer*.

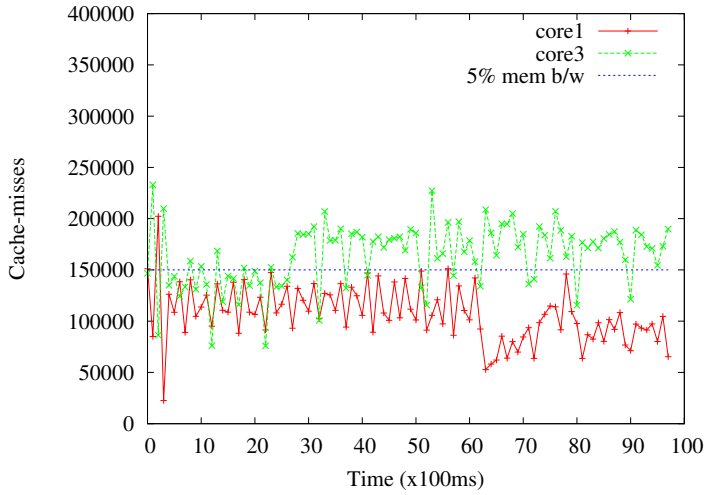
For static scheme, the total budget is configured to be evenly divided between the two cores. We set the period equal to 10ms and the budget equal to approximately 5% of memory bandwidth (15000 cache-misses per 10ms per core). For dynamic configuration, we set the period equal to the same 10ms and the shared global budget equal to approximately 10% of memory bandwidth (30000 cache-misses per 10ms on both cores).

Figure 4.7 shows measured cache-misses behavior of static budget distribution scheme and dynamic budget distribution scheme. X-axis is time in 100ms unit and Y-axis is the number of measured cache-misses for each 100ms time interval. At any time instance, both schemes limit the total number of cache-misses less than the global budget (300,000 misses for every 100ms). In the case of static scheme, both Core1 and Core3 are limited to the specified 5% bandwidth (150,000 misses) all the time. Note that, ideally both Core1 and Core3 should not exceed 5% bandwidth line. However, due to the limitation of current implementation, detailed in Section 4.5, we are observing small fluctuation. In case of dynamic scheme, each core can generate much more cache-misses than 5% bandwidth line, but when combines the cache-misses for both cores, it does not exceed 10% bandwidth limit. This allows more efficient bandwidth utilization.

Table 4.1 compares total throttled time, the amount of suspended time by our kernel throttling controller, measured over ten seconds of the video



(a) Static budget distribution



(b) Dynamic budget distribution

Figure 4.7: Cache-miss differences between static vs dynamic budget distribution scheme. Red line is Core1 and blue line is Core3

playback experiment. In the static scheme, Core1 and Core3 are throttled total 9,689ms (Core1: 3019ms, Core3: 6670ms), while they are throttled only 4,622ms in the dynamic scheme. This is because of more efficient global budget distribution of the dynamic scheme.

4.4.5 Evaluation Summary

The evaluation results first demonstrate response time impact of memory throttling both analytically and experimentally on a real hardware platform. The results suggest that our technique can provide isolation guarantee with analytic support. We also investigate performance impact of tasks on the throttled cores under static and dynamic budget distribution schemes. The result shows that dynamic distribution scheme can provide better performance to the throttled cores for the same aggregated memory bandwidth budget. However, this can adversely affect to the critical core due to increased contention and poor analytic bound.

4.5 Discussion

There are several assumptions we made on hardware that may affect the validity of our analysis results. We assume CPU synchronously waits fetch for cache-miss while it may concurrently execute out-of-order instructions in reality. Also, we assume that each single cache-miss takes a constant time, but it can vary in most COTS DRAM controllers (e.g., DRAM access cost significantly differ whether data is located in an opened row or a closed row). Taking into account of memory access cost could be an interesting research topic. Finally, we assumed bus arbitration schemes follows round-robin. However, actual arbitration is not well known and may differ from vendors. Developing a sound analysis framework on top of COTS components is challenging future research topic. On the other hand, even though our evaluation platform may not be a perfect match with the analysis model, we found that our analysis still gives us useful information to understand and configure systems.

4.6 Related Works

Shared resource contention in modern multicore/multiprocessor systems is a big challenge in real-time system design. Much effort has been spent to develop analysis frameworks for shared resource arbitration, in particular bus and memory, to compute worst case timing bounds. Thiele et al. presented Real-time calculus [107, 108] to model real-time tasks with request and service curves. Real-time calculus is extended to support multiprocessor systems by Leontyev et al [109]. Pellizzoni et al, also used real-time calculus to model CPU memory traffic and PCI IO traffic [103]. Rosen et al, designed a TDMA based system bus arbiter with algorithms that produce efficient TDMA bus schedules [110]. Schranzhofer el al. developed an analysis framework to compute the worst-case response time of real-time tasks under TDMA based bus arbitration and adaptive arbitration [111, 102]. Pellizzoni el al. also developed a delay analysis framework for round robin and fifo arbitration based multiprocessor systems [5]. They assumed a task consists of a set of superblocks and described the method to compute tight WCET bound. Our delay model uses the main results from [5, 103] to compute maximum throttling parameters that still guarantees the schedulability of a critical core. Recently Dasari et al. [112] developed a response time analysis for COTS based multi-core systems. However, they do not consider specific bus arbitration scheme and task cache-miss behavior, hence more pessimistic than our analysis.

Better timing guarantees on accessing shared memory and bus can be achieved by adopting specially designed hardware architectures. Several researchers proposed predictable DRAM controllers [48, 51] that employ predictable arbitration/access schemes to provide bandwidth and latency guarantees. Our analysis strives to utilize these hardware supported components, but provide flexibility in to achieve desired performance goal with the help of software assisted memory access control system.

In real-time systems, various aperiodic real-time servers, such as deferrable server, polling server, and sporadic servers, are used execute aperiodic tasks together with other real-time tasks without messing schedulability analysis [36]. Deferrable server maintains the budget for the duration of each period. While it maximally utilizes the given budget, it lowers schedulability because the budget can be executed in sequence spanning two consecutive

periods. Due to its simplicity in implementation, however, it is commonly used in practice as shown in CPU bandwidth controller in recent Linux 3.2 kernel [113]. Sporadic server is theoretically best but complex to implement and suffer significant overhead in high load [114]. Our memory throttling controller implementation follows the semantic of deferrable server.

4.7 Summary

In this chapter, we presented a response time analysis method under memory bandwidth controlled multicore systems. In particular, we considered a scenario where a dedicated core executes critical tasks while other cores execute tasks which cause significant accesses on the shared bus and memory. We presented a software based memory throttling mechanism and analytic solutions to compute throttling parameters that guarantee schedulability of critical tasks while minimizing performance impact of tasks on the throttled cores. We implemented the mechanism in Linux kernel and experimentally demonstrated the viability of our approach.

CHAPTER 5

ENERGY OPTIMIZATION

Another challenging issue for a RTOS is managing power/energy consumption of the system without violating required real-time performance. In this chapter, we describe a method to manage multiple dynamic voltage/frequency scaling (DVFS) capable components in a way to minimize energy consumption while still meeting timing requirements of real-time tasks.

DVFS schemes are common for reducing energy consumption, and many devices support multiple frequency and voltage levels. However, most DVFS schemes only adjust CPU frequency and voltage, and do not consider the energy consumption of the bus and memory. Previous studies show that bus and memory also significantly contribute to the total energy consumption [115]. Recent hardware allows these components to have their own clocks and DVFS capabilities that can be tuned independently of the CPU frequency. Therefore, new DVFS schemes must consider CPU, bus, and memory frequency to reduce the system-wide energy consumption.

Table 5.1 shows how a multiple component DVFS (multi-DVFS) scheme can save energy with a small performance penalty. We measured the energy consumption and execution time of two tasks — *dhrystone*, a CPU intensive task, and *memxfer5b*, a memory intensive task on an ARM926-ejs processor. For *dhrystone*, reducing memory frequency to half of the maximum increases the execution time by only 0.5% but reduces energy consumption by 10%. Lowering the CPU frequency to half of the maximum for *memxfer5b* results in a 2.6% increase in execution time but achieves a 30% energy reduction. The results show the potential of jointly adjusting CPU an memory frequencies/voltages in order to achieve best energy reduction.

In this chapter, we propose a multi-DVFS energy model that considers the energy consumption of CPU, bus, and memory, and considers task set characteristics such as the number of CPU and memory access cycles. We validate the model with a series of experiments on an ARM based embedded

Table 5.1: Effect of task characteristics in energy saving measured on a real hardware platform.

Task	CPU (MHz)	Mem (MHz)	Time (s)	Energy (mJ)	Energy savings
dhrystone	200	100	4.26	2364	–
	200	**50**	4.28	2106	10%
memxfer5b	200	100	3.46	1690	–
	100	100	3.55	1182	30%

system and show that it captures real system energy consumption. Based on our energy model, we present a systematic scheme for assigning multiple DVFS frequencies for a set of periodic real-time tasks given system and schedulability constraints. We show the effectiveness of the proposed scheme by both simulations and real experiments.

In summary, we make the following contributions: We propose a realistic multi-DVFS energy model that considers CPU, system bus, memory, and task set characteristics, at multiple frequency settings and validate it on a real hardware platform; based on the proposed model, we present a static multi-DVFS scheme for energy optimal scheduling of periodic real-time tasks.

This chapter is organized as follows: Section 5.1 presents the energy model and its validation on a real hardware platform. Section 5.2 reports model validation results on two real hardware configurations. Section 5.3 defines and solves the frequency assignment problem for a set of real-time tasks. Section 5.4 compares the proposed multi-DVFS scheme to other DVFS schemes, and Section 5.5 discusses practical issues. Section 5.6 concludes this chapter.

5.1 Energy Model

Most recent ARM based systems are capable of independently tuning CPU, system bus and memory frequencies [116, 117]. Our model incorporates independent frequency assignment and is validated on a real hardware platform. This section describes the multi-DVFS energy model that considers energy consumption on a platform with multiple independently adjustable component clocks. Table 5.2 presents a summary of notation used throughout the paper.

Table 5.2: Summary of notation.

E	total energy consumption (mJ)
E_{comp}	pure execution block energy consumption(mJ)
E_{mem}	cache stall block energy consumption (mJ)
E_{idle}	idle block energy consumption (mJ)
e	execution time of a given task (s)
P	period(=deadline) of a given task (s)
C	CPU cycles of a given task (10^6 cycles)
M	memory cycles of a given task (10^6 cycles)
r	cache stall ratio
f_c	CPU clock (MHz)
f_b	system bus clock (MHz)
f_m	memory clock (MHz)
V_{cpu}	CPU voltage (V)
V_{bus}	system bus voltage (V)
V_{mem}	memory voltage (V)
I	idle time dynamic power consumption of CPU, bus, and memory (mW)
R	static power consumption of the system (mW)
K_{ca}	capacitance constant for active CPU (nF)
K_{cs}	capacitance constant for standby(on but idle) CPU (nF)
K_{ba}	capacitance constant for active system bus (nF)
K_{bs}	capacitance constant for standby system bus (nF)
K_{ma}	capacitance constant for active memory (nF)
K_{ms}	capacitance constant for standby memory (nF)

We propose an energy model to reflect the actual characteristics of recent embedded platforms by focusing on three components: CPU, system bus, and main memory. These components are tightly integrated and contribute significantly to the total energy consumption as shown in Table 5.1. We also incorporate task set characteristics, specifically the number of CPU and memory access cycles, into the energy model.

Fig. 5.1 illustrates our energy model for a single task. In the model, task execution time is split into three blocks: (1) pure execution, (2) cache stall, and (3) idle. In the pure execution block, the CPU core executes instructions while the system bus and main memory are in standby. In the cache stall block, the cache fetches data from memory through the system bus while

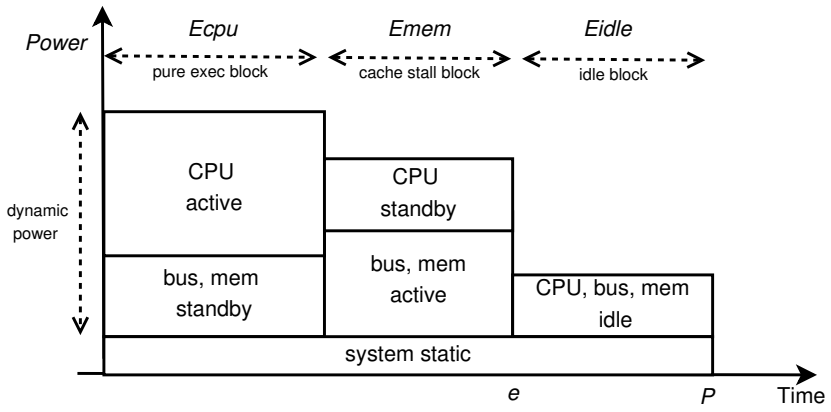


Figure 5.1: Energy model for a single task with deadline (e: task finish time, P: deadline).

the CPU core is in standby, waiting for the data to become available in its cache. After the task finishes, all three components – CPU, system bus, and memory – are in an idle state. While actual cache stall periods are scattered throughout the entire execution, we aggregate them into a single block. This is valid since most embedded processors execute in-order and there is no overlap between execution and off-chip memory fetch operations. For out-of-order processors, there is an overlap period, but it is relatively small because off-chip memory access takes much longer than executing out-of-order instructions.

Hence, the task execution time can be expressed as

$$e = \frac{C}{f_c} + \frac{M}{f_m}, \tag{5.1}$$

where C is the number of CPU cycles needed to complete the task and M is the number of memory cycles for off-chip memory access during cache stall handling; f_c is CPU clock frequency and f_m is main memory frequency. Using Eq. (5.1), we are able to predict the execution time of a task for a specific f_c and f_m. The first term, $\frac{C}{f_c}$, is the pure execution time (the first block in Fig. 5.1), and $\frac{M}{f_m}$ is the cache stall time (second block in Fig. 5.1). Idle time is the period minus the execution time: $P - e$. We define cache stall ratio as

$$r = \frac{M}{C + M}, \tag{5.2}$$

describing the CPU or memory intensiveness of a task.

The total energy consumption shown in Fig. 5.1 is expressed by

$$E = E_{comp} + E_{mem} + E_{idle}, \tag{5.3}$$

where E_{comp} is the system-wide energy consumption during the pure execution block, E_{mem} is the consumption during the cache stall block and E_{idle} is the consumption during the idle block. The total power consumption at any given time can be expressed as the sum of each component's power consumption. The power consumption of each component can be described using a well-known power equation, $K \cdot V^N \cdot f + R$, where K is half of the average capacitance, V is voltage, f is frequency of the component, N is the voltage exponent, and R is static leakage power [118]. Note that K is unique to each mode of operation. For example, during idle and cache stall time, the CPU consumes less power than when it is actively executing instructions, although its operating frequency and voltage may remain the same. The value of K for each operation mode may vary greatly for different processors. To generalize the model, we consider three modes – active, standby, and idle – and use multiple K values in our base equation. Other components – LCD, flash, etc. – consume static power regardless of the voltages and frequencies of the CPU, bus and memory. We do not consider a dynamic on/off strategy for those devices. In view of the above, we can write

$$E_{comp} = (K_{ca} \cdot V_{cpu}^{N_1} \cdot f_c + K_{bs} \cdot V_{bus}^{N_2} \cdot f_b + K_{ms} \cdot V_{mem}^{N_3} \cdot f_m + R) \cdot \frac{C}{f_c}. \tag{5.4}$$

Eq. (5.4) shows the energy consumption for the pure computation block. In this block, the CPU is actively executing instructions while the system bus and memory are in standby. K_{ca} is the capacitance constant for the active CPU mode. K_{bs} and K_{ms} are standby capacitance constants for system bus and memory, respectively. R represents the static power consumption of the entire system. Similarly,

$$E_{mem} = (K_{cs} \cdot V_{cpu}^{N_1} \cdot f_c + K_{ba} \cdot V_{bus}^{N_2} \cdot f_b + K_{ma} \cdot V_{mem}^{N_3} \cdot f_m + R) \cdot \frac{M}{f_m}. \tag{5.5}$$

Eq. (5.5) is the energy consumption during the cache stall block in which

the CPU stalls the execution and waits until the data becomes available in its cache. The CPU consumes less power when it is waiting for the cache data due to clock gating technology [119], so we introduce a constant factor, K_{cs}, for CPU standby mode. Both system bus and memory are active in this phase. K_{ba} and K_{ma} denote capacitance constants of active mode bus and memory, respectively. Finally,

$$E_{idle} = (I + R)\cdot(P - e). \tag{5.6}$$

Eq. (5.6) is the energy consumption during idle mode. Many recent embedded processors support a special idle mode which significantly reduces power consumption [120], so we use a separate term, I, to represent the idle mode power consumption of the CPU, system bus, and memory.

5.2 Model Validation

In this section, we present validation results that demonstrate the accuracy of the model (Section 5.1) in predicting the energy consumption of an embedded hardware platform with an ARM926-ejs based processor [121]. On the same platform, we verified two different memory configurations – internal SRAM (Section 5.2.1) and external DRAM (Section 5.2.2) – to show the application of the model to different devices.

5.2.1 SRAM Configuration

Fig. 5.2 shows the block diagram of the tested STMP3650 SoC. The SoC includes an ARM CPU core, L1 cache, system bus, and internal SRAM in a single package. The 256 KB internal SRAM is connected to the system bus.

While the proposed model is applicable to many embedded systems, there are several simplifications on our test platform; CPU, system bus, and internal SRAM all share the same power source ($V_{cpu} = V_{bus} = V_{mem}$) and system bus and memory operate at the same frequency ($f_b = f_m$). The resulting energy equation for our hardware platform is,

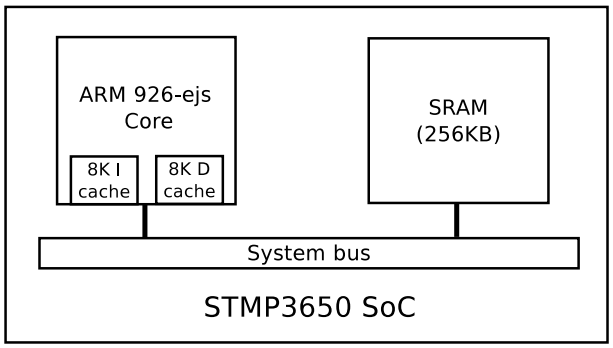


Figure 5.2: Tested hardware platform for SRAM configuration. The CPU, system bus, and SRAM share a common voltage. The SRAM operates at system bus frequency.

$$\begin{aligned} E = (K_{ca} \cdot V^N \cdot f_c + K^*_{ms} \cdot V^N \cdot f_m + R) \cdot \frac{C}{f_c} \\ +(K_{cs} \cdot V^N \cdot f_c + K^*_{ma} \cdot V^N \cdot f_m + R) \cdot \frac{M}{f_m} \\ +(I + R) \cdot (P - e). \end{aligned} \tag{5.7}$$

Because system bus and memory share a common voltage and frequency, we combine the terms for these components in Eq. (5.4) and (5.5), and use combined capacitance constants K^*_{ms} and K^*_{ma} to denote standby and active mode, respectively. These restrictions are platform and architecture specific. For example, [122] does not share a common voltage between the CPU and system bus. Table 5.3 shows the basic specifications of the tested processor. CPU frequency, f_c, is adjustable from 20 MHz to 200 MHz, the system bus and memory clock is divided from the CPU clock, $f_b = f_m = f_c/n$ where n is an integer, and voltage can be adjusted with 0.32 V steps from 1.504 V to 1.824 V. We set the voltage proportional to the CPU clock based on the recommendation of the processor data sheet, $V = af_c + b$. For this system $a = 0.0016$ V/MHz and $b = 1.504$ V.

In our experiments, energy consumption was measured for four synthetic tasks with different cache stall ratios, 0%, 10%, 25%, and 55%. We measured the entire board level power consumption which measures the power between the external supply and the input to the board. Our energy model requires the number of CPU cycles, C, and memory cycles, M to be known for a task.

Table 5.3: Processor specifications.

CPU clock	20 - 200 MHz (2 MHz step)
Bus clock	20 - 100 MHz (f_c/n)
Voltage	1.504 - 1.824 V (0.32 V step)
L1 cache	8 KB I, 8 KB D

Table 5.4: Model parameters for the tested hardware.

Capacitance (nF)				Power (mW)	
K_{ca}	K_{cs}	K^*_{ma}	K^*_{ms}	I	R
0.51	0.22	0.54	0.21	6.57	67.43

Note that M is the number of memory cycles to handle cache-misses; cache hit memory references are not included in M. In many recent processors, the cache stall cycles can be obtained by using a performance counter [123]; however, the tested processor did not have a counter. Therefore, we devised a program with two loops – a loop with 100% cache misses and another with 100% cache hits – and measured their execution time. To construct a loop with 100% cache misses, we allocated an array twice the size of the cache and sequentially read words separated by one cache-line size. These reads always resulted in cache misses. By subtracting the execution time of the 100% cache-hit loop from the 100% cache-miss loop, we obtained cache stall time.

By varying the number of loop iterations, we synthesized tasks with different C and M values. The instruction code of the synthetic tasks fit into the 8 KB I-cache to avoid generating additional instruction fetch cache stalls. For each task, eight different frequency/voltage settings were tested, and for each setting we measured energy consumption and execution time. We set the voltage exponent, $N = 2$, and performed nonlinear least squares analysis on the collected data to determine the value of each parameter in Eq. (5.7).

Fig. 5.3 plots the measured energy consumption and the model predicted energy consumption. The R^2 (the coefficient of determination that estimates the validity of a model) value is 99.97% and the maximum relative error is less than 2%, suggesting that our energy model accurately captures system

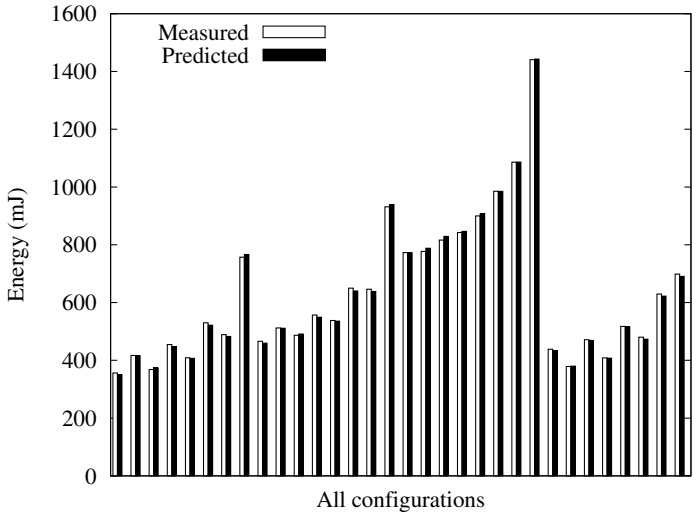


Figure 5.3: Energy model fitting for SRAM configuration. Comparison of measured and model predicted energy values for 32 configurations with varying cache stall ratio and clock settings. R^2 is 99.97%.

behavior. Table 5.4 shows the value of each parameter in Eq. (5.7) for the tested hardware.

5.2.2 DRAM Configuration

We changed the hardware configuration of Section 5.2.1 to use external DRAM. Fig. 5.4 shows the modified system. DRAM is connected to on-chip DRAM controller, which is attached to system bus. One key difference from SRAM is that DRAM does not share a common voltage with the processor core, resulting in a different system energy equation.

In our platform, the DRAM [124] uses a fixed 3.0V and operates at a multiple of the system bus frequency. CPU and system bus use the same variable voltage, from 1.504V to 1.824V, as the previous section.

The energy equation of this hardware configuration is,

$$\begin{aligned} E = (K_{ca}{\cdot}V^N{\cdot}f_c + K^*_{ms}{\cdot}(V^N{\cdot} + 3.0^N){\cdot}f_m + R){\cdot}\frac{C}{f_c} \\ +(K_{cs}{\cdot}V^N{\cdot}f_c + K^*_{ma}{\cdot}(V^N{\cdot} + 3.0^N){\cdot}f_m + R){\cdot}\frac{M}{f_m} \\ +(I+R){\cdot}(P-e). \end{aligned} \tag{5.8}$$

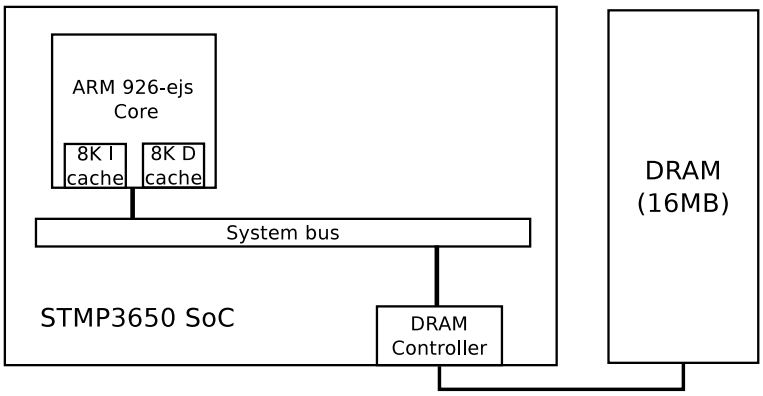


Figure 5.4: Tested hardware platform for DRAM configuration. DRAM uses a fixed voltage (3.0V) while CPU and system bus share a common varying voltage. The DRAM operates at system bus frequency.

The K^*_{ma} and K^*_{ms} represent the combined capacitance of active and standby bus and DRAM, similar to Section 5.2.1. However, we use two voltage terms, $V^N + 3.0^N$, because the system bus and DRAM do not share a common voltage, only a common frequency, f_m. We set $V = 0.0016 f_c + 1.504$ and use $N = 2$.

We measured the energy consumption of four synthetic task configurations with varying cache stall ratios – 0%, 10%, 25%, and 55% – similar to the experiments in Section 5.2.1. However, all cache stalls are for DRAM; there are no SRAM accesses. For each task configuration, eight different clock-voltage settings were tested, and for each setting we measured the entire board level power consumption and execution time. As in the previous section, we performed a nonlinear least square analysis on the collected energy consumption data to determine the value of each parameter in Eq. 5.8.

Fig. 5.5 plots the measured energy consumption and the model predicted energy consumption. The R^2 (the coefficient of determination) value was 99.78% and the mean average error (MAE) was 1.25%, suggesting that our energy model accurately captures the system behavior. Table 5.4 shows the value of each parameter in Eq. (5.8) for the tested hardware configuration with external DRAM, obtained from the non-linear least square analysis. We used these parameters for the evaluation of real applications in Section 5.4.2.

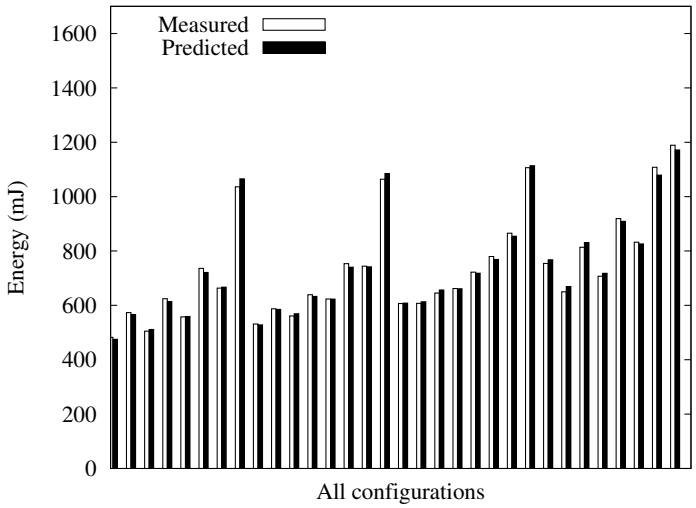


Figure 5.5: Energy model fitting for DRAM configuration. Comparison of measured and model predicted energy values for 32 configurations with varying cache stall ratio and clock settings. R^2 is 99.78%.

Table 5.5: Model parameters for the system with a external DRAM

Capacitance (nF)				Power (mW)	
K_{ca}	K_{cs}	K^*_{ma}	K^*_{ms}	I	R
0.52	0.30	0.18	0.05	6.52	71.18

5.3 Energy Optimization of Real-Time Tasks

We formulate a problem to find the energy optimal frequency set, which contains the frequency assignments of multiple DVFS components, to schedule periodic real-time tasks. In our solution, a single (possibly different) frequency is found for each component. All tasks that share a component, share that frequency on the component. We call such a frequency assignment static. In theory, it is possible to do better by implementing a dynamic frequency assignment that changes the frequency of each component on each context switch. The problem of dynamic frequency assignment becomes one of finding the best frequency for each component and for each task. We do not consider the latter problem, because it is difficult to implement in practice. Nevertheless, in the evaluation section, we compare the performance of our static scheme to the optimal dynamic solution (which we computed by a brute-force search for small task sets). We show that the dynamic scheme does not offer significant improvement in energy savings, further reinforcing our choice of a static (multi-DVFS) frequency assignment.

5.3.1 Problem Definition

Given a set $T = T_1, ..., T_n$ of n periodic real-time tasks, the period of T_i is denoted by P_i, which is equal to the deadline. The tasks are scheduled on a single processor system based on preemptive scheduling, and all tasks are assumed to be independent. In the worst case, each task invocation requires C_i CPU cycles and M_i memory cycles. The worst case execution time of task T_i is $e_i = \frac{C_i}{f_c} + \frac{M_i}{f_m}$, where f_c is CPU frequency, f_b is system bus frequency, and f_m is memory frequency (see Eq. (5.1)).

The energy consumption, $E_{act,i}$, of each invocation of task T_i is given by $E_{act,i} = E_{comp,i} + E_{mem,i}$ (see Eq. (5.4), (5.5)). CPU, system bus, and memory are idle if there is no task to execute, and the power consumption is $I + R$ (see Eq. (5.6)). The hyperperiod, H, is the least common multiple of $P_1, ..., P_n$, and the total energy consumption, E, over H is $\sum_{i=1}^{n} \frac{H}{P_i} \cdot E_{act,i} + E_{idle}$, where $E_{idle} = (H - \sum_{i=1}^{n} \frac{H}{P_i} \cdot e_i) \cdot (I + R)$. A schedule of periodic tasks is *feasible* if each task, T_i, is guaranteed C_i CPU cycles and M_i memory cycles at each invocation.

The optimization formulation of the multi-DVFS frequency assignment

problem with EDF scheduling is

$$\text{minimize} \quad \sum_{i=1}^{n} \frac{H}{P_i} E_{act,i} + E_{idle} \tag{5.9}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{e_i}{P_i} \leq 1. \tag{5.10}$$

Eq. (5.9) minimizes the sum of the energy consumption of all task invocations during hyperperiod H. Eq. (5.10) is the necessary and sufficient schedulability constraint because the utilization of task T_i is $\left(\frac{C_i}{f_c} + \frac{M_i}{f_m}\right)/P_i$.

5.3.2 Static Multi-DVFS Frequency Assignment

In this section, we present an optimal solution for assigning a single frequency set to a task set, T, such that all the tasks in the set run at the same CPU frequency, f_c, and the same memory frequency, f_m. The number of CPU execution cycles in a *hyperperiod*, C_H, is $\sum_{i=1}^{n}\left(\frac{H}{P_i} \cdot C_i\right)$, and the number of memory access cycles, M_H, is $\sum_{i=1}^{n}\left(\frac{H}{P_i} \cdot M_i\right)$ [1]. Total execution time in the *hyperperiod*, e_H, is $\frac{C_H}{f_c} + \frac{M_H}{f_m}$.

Lemma 1. (5.9) *is equivalent to the sum of the right hand side of* (5.4)–(5.6) *by replacing* C *with* C_H *and* M *with* M_H.

Proof. Let W_{comp} be the power during the pure execution block and W_{mem} be the power during the cache stall block. Thus we have

[1] While individual M_i, cache stall cycles of task T_i, can increase when preempted by other tasks, the effect the increase is generally negligible – 0.25% of the total execution time in maximum when preempted 100 times for 3 seconds on an ARM926-ejs processor [125]. Nevertheless, we can measure task set cache stall cycles, M_H, including additional stall cycles caused by preemption, using performance counter.

$$
\begin{aligned}
&\sum_{i=1}^{n} \frac{H}{P_i} E_{act,i} + E_{idle} \\
&= \sum_{i=1}^{n} (\frac{H}{P_i} W_{comp} \frac{C_i}{f_c} + \frac{H}{P_i} W_{mem} \frac{M_i}{f_m}) + E_{idle} \\
&= \frac{1}{f_c} \sum_{i=1}^{n} \frac{H \cdot C_i}{P_i} W_{comp} + \frac{1}{f_m} \sum_{i=1}^{n} \frac{H \cdot M_i}{P_i} W_{mem} + E_{idle} \\
&= W_{comp} \frac{C_H}{f_c} + W_{mem} \frac{M_H}{f_m} + E_{idle}.
\end{aligned}
\tag{5.11}
$$

□

Lemma 2. *Under EDF, if the execution time, e_H, does not exceed the hyperperiod, H, then the task set is schedulable.*

Proof. From Eq. (5.10),

$$
\sum_{i=1}^{n} \frac{e_i}{P_i} = \sum_{i=1}^{n} \frac{H \cdot e_i}{H \cdot P_i} = \frac{1}{H} \sum_{i=1}^{n} \frac{H}{P_i} \cdot e_i = \frac{e_H}{H} \leq 1. \tag{5.12}
$$

Hence,

$$
e_H \leq H. \tag{5.13}
$$

□

Methodology

The energy model presented in Eq. (5.11) can be combined with system and deadline constraints to find the energy optimal f_c, f_b, and f_m given a task set defined with a single hyperperiod, H.

Frequency constraints on the CPU, memory, and bus arise from hardware specifications,

$$
f_{c,min} \leq f_c \leq f_{c,max}, \tag{5.14}
$$

$$
f_{m,min} \leq f_m \leq f_{m,max}, \tag{5.15}
$$

$$
f_b = f_m. \tag{5.16}
$$

The procedure for determining the energy optimal frequency assignment requires finding unconstrained energy minimizing frequency sets and finding

solutions on the boundary conditions and the boundary intersections imposed by the constraints. Each frequency set must then be evaluated in the energy model and the energy minimal solution chosen.

The optimum frequency assignment is found assuming continuous variables, but real systems have discrete frequency steps. The frequency assignment for the real system can be found by testing possible frequencies that neighbor the continuous optimal frequency solution. Neighboring frequencies can be enumerated by finding the nearest higher and lower discrete frequencies. Frequency sets that violate the deadline constraint are eliminated and the remaining sets must be evaluated and compared in the energy model.

The energy model presented in Eq. (5.11) and the constraints in Eqs. (5.13)–(5.16) result in the following set of equations that find possible frequency assignments in the global search space and on the boundary conditions.

Unconstrained Minima

$$\text{Find } f_m \text{ such that } \frac{\partial E}{\partial f_m} = 0 \tag{5.17}$$

$$\text{Find } f_c \text{ such that } \frac{\partial E}{\partial f_c} = 0 \tag{5.18}$$

Substituting f_m from Eq. (5.17) into Eq. (5.18) yields unconstrained energy minimum frequency sets.

Minima on CPU Frequency Boundaries

$$\text{Find } f_m \text{ such that } \frac{\partial E}{\partial f_m} = 0 \tag{5.19}$$

$$f_c \in \{f_{c,min}, f_{c,max}\} \tag{5.20}$$

Eq. (5.19) solved for the scenarios where $f_c = f_{c,max}$ and $f_c = f_{c,min}$ yields the energy minimum frequency sets on the CPU frequency boundaries.

Minima on Memory Frequency Boundaries

$$f_m \in \{f_{m,min}, f_{m,max}\} \tag{5.21}$$

$$\text{Find } f_c \text{ such that } \frac{\partial E}{\partial f_c} = 0 \tag{5.22}$$

Eq. (5.22) solved for the two scenarios where $f_m = f_{m,max}$ and $f_m = f_{m,min}$ yields the energy minimum frequency sets on the memory frequency boundaries.

Minimum on Deadline Constraint Boundary In order to meet the task set deadline, H,

$$f_m = \frac{M_H}{H - \frac{C_H}{f_c}} \tag{5.23}$$

$$\text{Find } f_c \text{ such that } \frac{\partial E}{\partial f_c} = 0 \tag{5.24}$$

Substituting f_m from Eq. (5.23) into Eq. (5.24) yields the energy minimum frequency set on the deadline constraint boundary.

Boundary Intersections All combinations of maximum and minimum CPU and memory frequencies yield the frequency sets at frequency boundary intersections. Eq. (5.23) solved for the scenarios where $f_c = f_{c,max}$ and $f_c = f_{c,min}$, and backsolved for f_c when $f_m = f_{m,max}$ and $f_m = f_{m,min}$ yields the frequency sets on the deadline and frequency boundary intersections.

Methodology Summary

The following steps summarize the procedure for finding multiple component frequency assignments:

1 Find unconstrained energy optimal frequency sets using Eqs. (5.17)–(5.18) and the energy optimal frequency sets on each boundary condition and boundary intersection using Eqs. (5.19)–(5.24).

2 Eliminate results that violate any constraints from Eqs. (5.13)–(5.16).

3 Evaluate and compare each frequency set in the energy model from Eq. (5.11), and choose the lowest energy set.

4 Enumerate the frequency sets obtainable in the real system that neighbor the optimal frequency set.

5 Eliminate sets that violate the deadline constraint, Eq. (5.13), and evaluate the remaining frequency sets in the energy model from Eq. (5.11), choosing the lowest energy set as the final solution.

An example of applying the procedure is presented below.

Methodology Example

Consider a system with $V_{cpu} = V_{bus} = V_{mem}$ and CPU voltage as a linear function of f_c, $V_{cpu} = A\,f_c + B$. Let $A = 0.0016$ V/MHz and $B = 1.504$ V. Other system parameters are given in Table 5.4. Let the task set be described by $C_H = 140 \cdot 10^6$ cycles, $M_H = 30 \cdot 10^6$ cycles, and hyperperiod, $H = 3$ sec. Fig. 5.6 shows the energy consumption of the task set as a function of f_c and f_m. No unconstrained minima are found within the limits of f_c. The lowest energy frequency set from all boundaries and boundary intersections is found on the deadline boundary at $\{f_c, f_m\} = \{65.45$ MHz, 35.35 MHz$\}$. It can be seen from Fig. 5.6 that this frequency assignment indeed results in the minimum energy consumption for the system. The neighboring obtainable frequency sets are {66,36}, {66,34}, {64,36}, and {64,34} if f_c and f_m are adjustable in 2 MHz steps. Checking the frequency sets against the constraints reveals that all sets except {66,36} violate the deadline constraint, so they are eliminated. Evaluating the remaining frequency set in the energy model in Eq. (5.11) determines that the set $\{f_c, f_m\} = \{66,36\}$ has an energy value of 501.3 mJ for this task set.

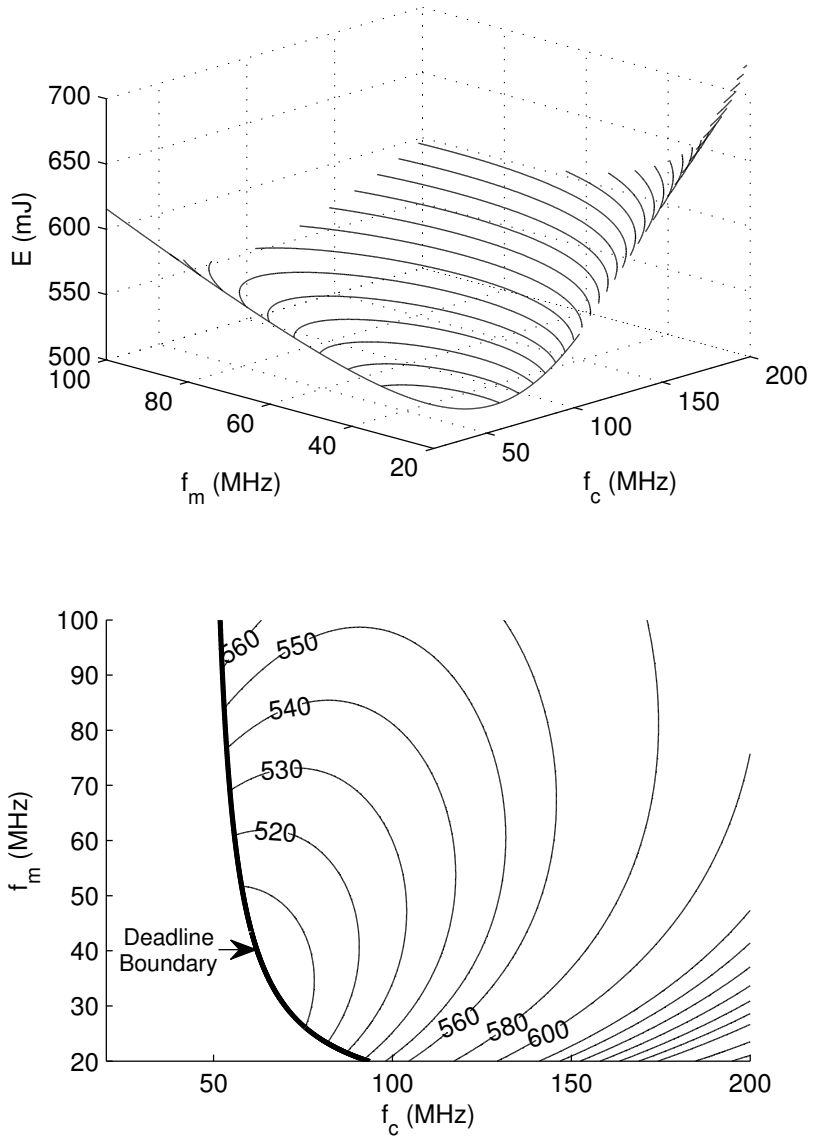


Figure 5.6: Energy vs. f_c and f_m with a task set of $C_H = 140 \cdot 10^6$ cycles, $M_H = 30 \cdot 10^6$ cycles and $H = 3$ sec evaluated in the proposed energy model.

5.4 Evaluation

In this section, we present simulation and experimental results performed on the hardware platform to evaluate the multi-DVFS scheme. In Section 5.4.1, we demonstrate that our multi-DVFS scheme can save energy compared to traditional CPU-only DVFS scheme and justify our choice of static multi-DVFS over dynamic multi-DVFS with simulations. In Section 5.4.2, we show that the multi-DVFS scheme saves energy with experiments that run real applications on the previously described hardware.

5.4.1 Simulation

In this section we simulated the energy savings of the proposed multi-DVFS scheme with other DVFS schemes. We performed simulations because (1) the choice of f_m is limited in our real hardware platform – $f_m = f_c/n$ where n is an integer – but many new processors [116, 117] do not have this constraint, and (2) we wanted to investigate the effects of varying the idle power and voltage range, which are fixed in the real hardware platform. Nevertheless, we used the parameters obtained from the real hardware platform as shown in Table 5.4. We use average power consumption, which is calculated by the total energy consumption divided by the *hyperperiod*, as the evaluation metric.

Five schemes are compared in our evaluation: *MAX*, *CPU-only DVFS*, *Baseline multi-DVFS*, *Static multi-DVFS*, and *Dynamic multi-DVFS*. In the *Max* scheme, tasks are executed with the maximum CPU and maximum bus/memory frequency. In the *CPU-only DVFS scheme*, we set the optimal static CPU frequency while the system bus/memory frequency is set to the maximum value. The *Baseline multi-DVFS scheme* requires that both the CPU and bus/memory frequencies be proportional to the task set CPU utilization at maximum frequency[1]. The *Static multi-DVFS scheme* is the proposed solution described in the previous section which assigns a single CPU and memory frequency to the entire task set. The *Dynamic multi-DVFS scheme* results from a brute-force search for all possible combinations of frequencies to determine the optimal frequencies for each individual task.

[1]In the *Baseline multi-DVFS* scheme, $f_c = f_{c,max} \cdot \sum_{i=1}^{n} u_{i,max}$, and $f_m = f_{m,max} \cdot \sum_{i=1}^{n} u_{i,max}$, where $u_{i,max} = \left(\frac{C_i}{f_{c,max}} + \frac{M_i}{f_{m,max}}\right)/P_i$.

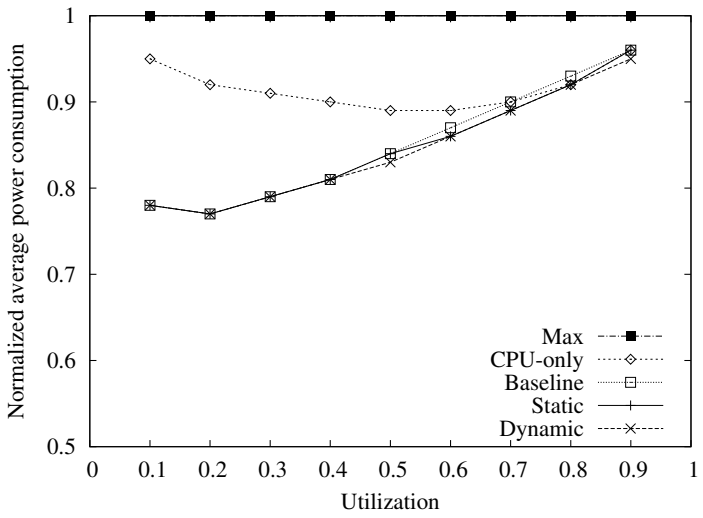


Figure 5.7: Average power consumption with varying utilization and constant cache stall ratio = 0.3.

We normalize the average power consumption to the *Max* scheme (*i.e.*, no DVFS) in every figure.

The system energy consumption was simulated by varying the task set utilization at maximum frequency, cache stall ratio, and idle power consumption while satisfying schedulability constraints for Earliest Deadline First (EDF) scheduling.

Varying Task Set Utilization

Fig. 5.7 shows the average power consumption of the compared DVFS schemes with varying utilization. As the utilization increases, the feasible frequency scaling range decreases; as a result, the effectiveness of all the DVFS schemes is reduced. When utilization is low, the *Static multi-DVFS* scheme consumes less energy than the *CPU-only DVFS* scheme, because it saves energy by setting a lower bus frequency without violating the deadline constraints. Although the *Baseline multi-DVFS* scheme is close to *Static* and *Dynamic optimum multi-DVFS* schemes in this figure, the effectiveness of the *Baseline multi-DVFS* highly depends on the cache stall ratio, which we will show in the next subsection. The difference in energy consumption between the *Static multi-DVFS* scheme and the *Dynamic multi-DVFS* scheme is less than 1%.

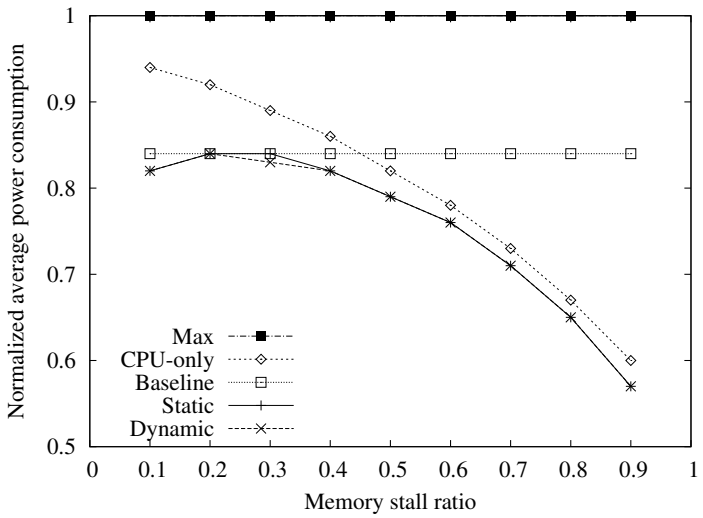


Figure 5.8: Average power consumption with varying cache stall ratio and constant utilization = 0.5.

Varying Cache Stall Ratio

Fig. 5.8 shows the average power consumption of the compared DVFS schemes for varying task set cache stall ratio, $\frac{M_H}{C_H+M_H}$. The task set utilization is fixed at 50%. When the cache stall ratio is low (representing a CPU intensive workload), the *Static multi-DVFS* scheme takes advantage of lowering the bus frequency without violating the deadline constraints. In contrast, when the cache stall ratio is high, the *CPU-only DVFS* and the *Static multi-DVFS* schemes have lower energy consumption than the *Baseline multi-DVFS* scheme, because they are able to set a lower CPU frequency. When the cache stall ratio is between 0.1 and 0.2, which is common in many applications [103], the *Static multi-DVFS* scheme shows a clear advantage. Note that the *Static multi-DVFS* scheme shows similar performance to the *Dynamic multi-DVFS* scheme.

Varying Cache Stall Ratio Diversity

In the next experiment, we change the degree to which different tasks differ in their cache stall ratio. Note that, when tasks are more diverse (i.e., when a mix of CPU intensive and memory intensive tasks are present), the *Static multi-DVFS* scheme is expected to perform worse than the *Dynamic multi-*

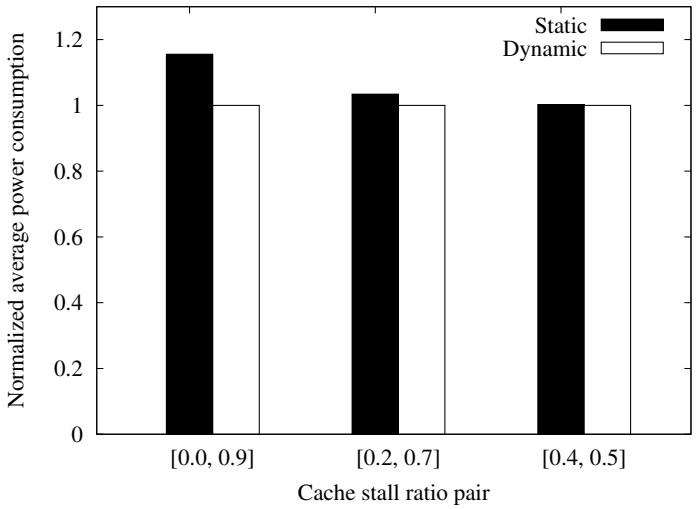


Figure 5.9: Comparisons on the average power consumption with different diversity of cache stall ratio and utilization = 0.5, task set cache stall ratio = 0.45. The configuration [*min*, *max*] represents a task set in which half of the tasks have a cache stall ratio of *min* and half have the ratio *max*.

DVFS scheme because it cannot customize frequency settings to each task. As a proxy for task diversity, we change the variance of the memory stall ratio across the task set keeping the task set cache stall ratio fixed at 0.45. In Fig. 5.9, the average power consumption is plotted against the variance in the cache stall ratio.

Note that, while the *Dynamic multi-DVFS* scheme is 13% better than the *Static multi-DVFS* scheme for large variances, it is expected that in many realistic embedded task sets tasks are reasonably homogeneous. For example, in a process control system, where different tasks implement different controllers, it is likely that the controllers do not substantially differ in memory stall ratio. For such sets, the performance hit of the *Static multi-DVFS* scheme is less than 0.5%, which we deem acceptable. Investigation of efficient dynamic multi-DVFS schemes is therefore left as a topic for future work.

Varying Voltage Scaling Range

Fig. 5.10 shows the average power consumption of the compared DVFS schemes over varying voltage scaling range. The cache stall ratio is 10% and the task set utilization is 50%. If the system provides a larger voltage

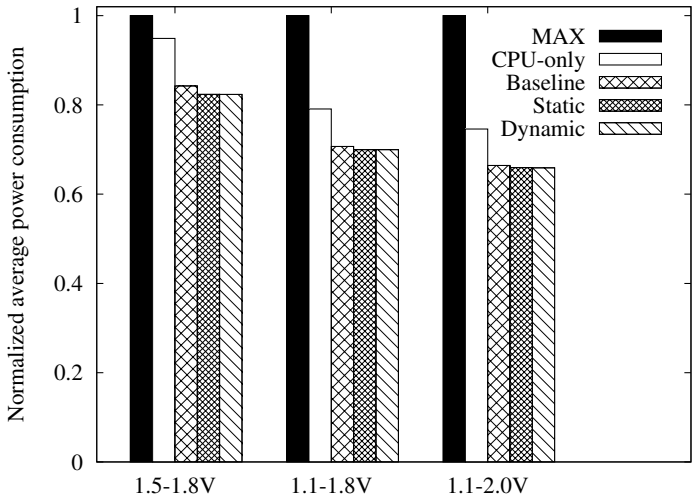


Figure 5.10: Comparisons on the average power consumption with different voltage scaling range and utilization = 0.5, cache stall ratio = 0.1.

scaling range, all DVFS schemes save more energy. However, all *multi-DVFS* schemes are able to take greater advantage of the larger scaling range by also controlling the bus frequency.

Simulation Summary

Simulation results show that cache stall ratio is one of the most important factors affecting the performance of the *Static multi-DVFS* scheme. The *Static multi-DVFS* scheme is effective for workloads with low cache stall ratio, because it is able to set a low memory frequency without violating deadline constraints. Additionally, the *Static multi-DVFS* scheme has lower energy consumption compared to the *CPU-only DVFS* scheme when idle power is high, because it can lower both the CPU and memory frequency and make the utilization close to unity. The proposed *Static multi-DVFS* scheme achieves good performance in a large range of evaluated configurations.

5.4.2 Experiments with Real Applications

In this section, we evaluate the multi-DVFS scheme on a real hardware using two applications: *madplay*, a mp3 decoder, and *dhrystone*, a performance

benchmark. The hardware configurations and the energy model are described in Section 5.2.2.

Obtaining C and M

Our energy optimization method requires knowing C and M, the CPU and memory cycles, respectively. Previously, we controlled C and M directly by changing the number of loops in our synthetic program (Section 5.2.1), however, there is no direct method to obtain these values for an arbitrary program without hardware support (e.g. a performance counter).

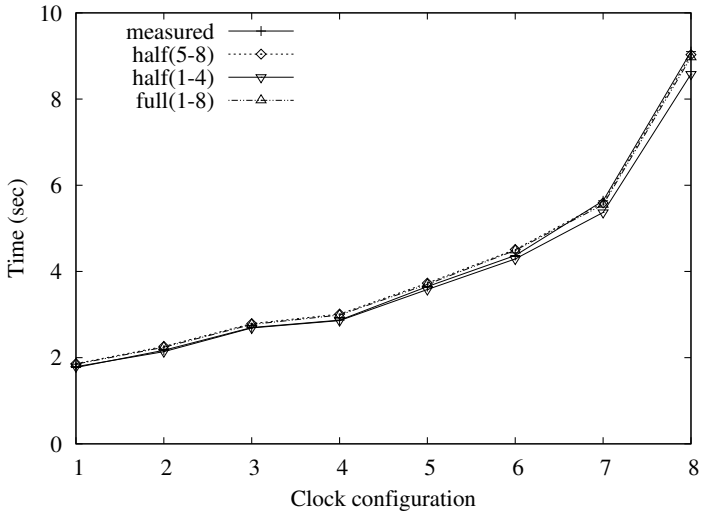
Instead, we obtained C and M values indirectly with regression analysis using the following methodology: (1) measure the execution time for various clock configurations (combinations of CPU and bus frequencies), and then (2) perform a non-linear least square analysis for Eq. (5.1) using the collected execution time data.

Fig. 5.11 shows the execution time regression results for *madplay* and *dhrystone* [2]. In addition, a partial data crosscheck was performed on the regression to verify its predictive capability. The full(1-8) data set used all eight collected data points for the regression; half(1-4) used four data points; half(5-8) used the complement of half(1-4). Table 5.11(c) compares mean absolute error (MAE) of each regression model suggesting that the predictive capability of the model is reasonable. The MAE of *dhrystone* is reasonably small, indicating that this method is effective for estimating the execution time of a CPU intensive task. The MAE of *madplay* is relatively large – up to 3%. We conjecture that frequent I/O operations (e.g. mp3 file reads) and the complexity of DRAM accesses result in this error.

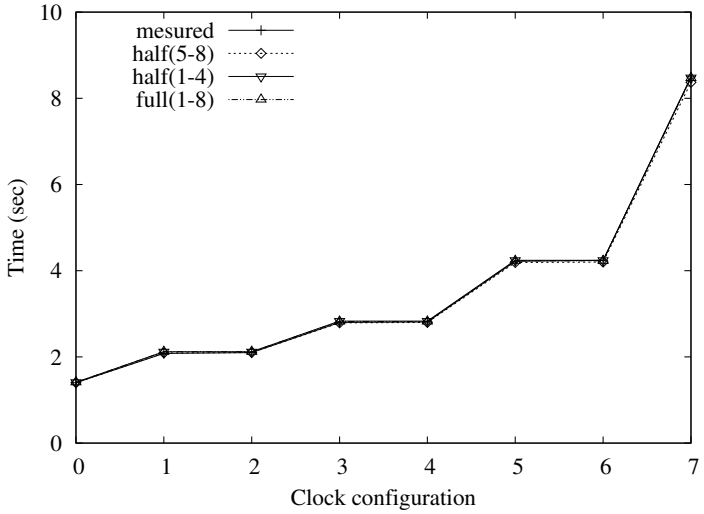
Energy Optimal Frequency Selection

From the obtained C and M values, we performed an experiment to compare the actual energy saving of our multi-DVFS scheme. We used a taskset that consisted of *madplay* and *dhrystone* with a deadline of 10 seconds in both programs. We used (C, M) values obtained from the regression: *madplay* is

[2]For *madplay*, we decoded a 10 second length, 1KHz sine wave mp3 file. Note that we outputted the decoded pcm data to `/dev/null` instead of `/dev/dsp`, because we did not consider the power consumption of the audio amplifier and DAC convertor chip in our energy model. For *dhrystone*, we used an input value of 100000.



(a) *madplay*



(b) *dhrystone*

data set	*madplay*	*dhrystone*
full(1-8)	2.72%	0.69%
half(1-4)	2.18%	0.79%
half(5-8)	3.08%	0.57%

(c) Mean Absolute Error

Figure 5.11: Execution time regression. measured shows actual measured execution time at eight different clock configurations. full(1-8), half(1-4), and half(5-8)) are regression models that used all eight data, first four data, and last four data respectively. Table (c) is MAE of each regression on both programs.

	MAX	CPU-only	Static	Dynamic
Freq. (MHz)	(200,100)	(100,100)	(40,20)	[(24,24),(66,22)]
Energy (mJ)	1701.00	1588.63	1403.63	1392.62
Saving	-	6.58%	17.46%	18.11%

Table 5.6: Comparison of actual energy saving. The second row, Freq. (MHz), is shown in (f_c, f_m) format. The last column, Dynamic, consists of two clock settings for *madplay* and *dhrystone*, respectively.

```
        ...
        // computation loop
cloop:
        mov     r0, r2                  // no cache access
        add     r2, r2, r1
        cmp     r2, r3
        blt     cloop
        ...
        // cache stall loop
mloop:
        ldr     r0, [ip, r2, asl #2]    // 100% cache miss
        add     r2, r2, r1              // increment pointer
                                        // to next cache line.
        cmp     r2, r3
        blt     mloop
```

Figure 5.12: Synthetic program for calibration

(137.09, 42.37) and *dhrystone* is (169.37, 0.00), and compared the energy consumption of the four schemes – *MAX*, *CPU-only DVFS*, *Static multi-DVFS*, and *Dynamic multi-DVFS* – as described in Section 5.4.1. The experiment was performed on the hardware platform described in Section 5.2.2.

Table 5.6 shows the actual energy saving for various schemes. The results show that multi-DVFS schemes (Static and Dynamic) outperform *CPU-only DVFS* scheme, as predicted from simulations, and the energy savings are up to 17% compared to *MAX*.

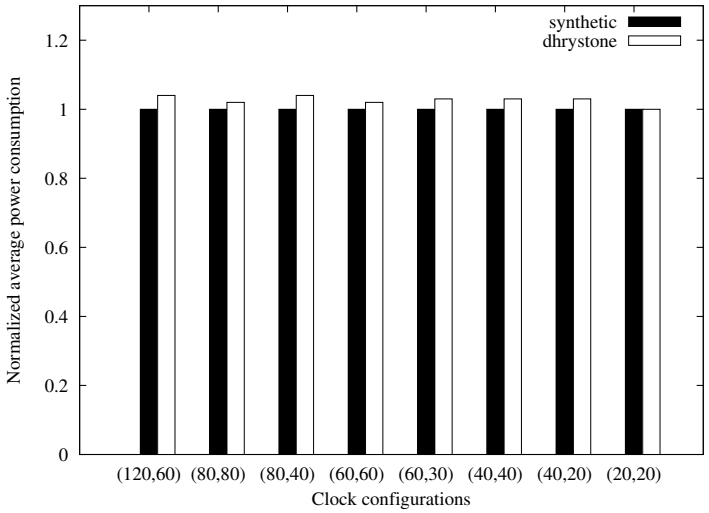


Figure 5.13: Comparison of average power consumption of *synthetic* and *dhrystone*.

5.5 Discussion

In this section, we discuss the validity of our energy model calibration method described in Section 5.1. Fig. 5.12 shows the main body of the synthetic program we used. The program consists of two loops: (1) the computation loop with no cache misses, and (2) the memory loop with 100% cache misses. By changing the repetition count of each loop, we synthesized multiple tasks with varying C and M values, and then used those tasks to calibrate our energy model. Notice that this program used only five instructions: `MOV`, `ADD`, `CMP`, `BLT`, and `LDR`. However, other instructions (e.g. `MUL`) may have different power consumption per cycle [126]. Also, while we include cache-hit memory access instructions (e.g. `LDR`) on cached data, in computation cycles, it may consume more power than typical computation instructions [127].

To investigate the effect of varying power consumptions for different instructions, we compared power consumption of our synthetic program, *synthetic*, and *dhrystone*. *dhrystone* generates almost no cache misses so we configured the synthetic program to only use the computation loop for comparison. Fig. 5.13 shows the average power consumption of the two programs at eight different clock configurations. The result is normalized to the synthetic program values. It shows that *dhrystone* consumes slightly more power

(up to 4%) than *synthetic*, meaning that the model predicted power consumption, calibrated from a set of *synthetic* programs, may be different from real power consumption depending on the application's instructions and the number of cache-hit memory accesses. One way to improve the accuracy of the power model is to extend it to include instruction and cache level power consumption as in [128]. However, accounting for instruction-level details will significantly complicate the model.

Minimizing the error while maintaining the simplicity of the model is a challenging task in general. A possible method to improve accuracy is to use representative real applications for model calibration. To do this, we would need a precise method to obtain computation and memory cycles for each application, for example, using a hardware performance counter as in [81].

5.6 Summary

In this chapter, we contribute to a realistic and flexible energy model for embedded systems. The model focuses on CPU, system bus, and memory and allows variable frequencies for those components. Through experiments on a real hardware platform, we showed that the model can accurately predict system-wide energy consumption. A solution was then derived to find frequency assignments for multiple components considering system constraints. Based on the model and the solution, we proposed a static multi-DVFS scheme to schedule periodic real-time tasks, and we compared our multi-DVFS scheme with other DVFS schemes to demonstrate its effectiveness.

CHAPTER 6

CONCLUSION

In this thesis, we have attempted to address two critical challenges in the design of real-time operating system for modern embedded processors: providing performance isolation and reducing power consumption.

We first focused on designing OS level mechanisms to improve isolation performance on accessing shared memory subsystem in commodity multicore platforms. The basic goal is to provide a memory bandwidth guarantee to each individual core in a given multicore platform where all cores share a memory subsystem. This is challenging because of the complexities of DRAM and commodity DRAM controllers. We have designed a software framework in Linux, called MemGuard, which is able to provide memory bandwidth guarantees for real-time tasks at the granularity of every OS tick, leveraging hardware performance counters. We also proposed reclaim and sharing algorithms that improve the overall memory bandwidth utilization.

Next, we have attempted to develop a worst-case response time analysis method for memory bandwidth regulated systems. Due to the seer complexities of commodity hardware components, developing a sound analysis framework is challenging. The analysis presented in this thesis, while limited, offers a meaningful insight in understanding and designing more predictable real-time systems.

Finally, we addressed the problem of finding best frequency assignments of multiple tightly coupled DVFS components for periodic real-time systems. By modeling and validating an energy model using a real embedded hardware platform, we demonstrated the feasibility of jointly adjusting multiple DVFS capable components. The proposed solutions were evaluated by simulation and experiments on an actual hardware platform.

6.1 Future Work

The research presented in this thesis is ongoing and we would like to pursue several interesting avenues for future research.

First, in terms of shared resource management, another promising avenue is DRAM aware memory allocation. Because DRAM is composed of multiple banks that can be accessed in parallel, as explained earlier, careful allocations can significantly reduce contention. For example, one may allocate memory for a core in a certain core-private memory bank, which would substantially reduce memory contention as shown in a recent study implementing private banking at the level of the DRAM controller design [50]. However, we believe OS level memory allocations would be more effective and flexible as it allows more fine-grain (page granularity) control compared to hardware based DRAM controller level approaches. We have already implemented a prototype DRAM-aware memory allocator in Linux kernel that shows promising early results. We plan to explore algorithms that provide better predictability without significantly reducing accessible memory space, which is the main problem of private banking based DRAM controllers [50, 53], for real-time systems. It would be also interesting to investigate joint application of the memory management techniques (both bandwidth and space) and cache space management techniques (e.g., [30]) for better predictability and performance.

Next, for emerging real-time systems such as Drones (or UAVs), their utility is largely limited by battery life as it determines how long they can operate in the air. Power-conscious scheduling is, therefore, of paramount importance. In this thesis, we presented the MultiDVFS that jointly optimizes multiple DVFS capable components for maximum energy saving. However, recent trends in micro-processor design suggests diminishing returns of DVFS and growing importance of DPM [129], because the relative contribution of static power consumption, which can be altered by DPM, has grown significantly compared to the contribution of dynamic power consumption, which can be adjusted by DVFS. We plan to extend the presented MultiDVFS work to consider multi-level sleep states offered by modern processors at the RTOS level for maximum energy saving for real-time systems.

Finally, we plan to investigate ways to provide better performance guarantees in the cloud computing infrastructure such as Amazon EC2. As reported

in [130], today's cloud platforms do not provide guaranteed computing performance: performance varies considerably over time and over different VMs of the same type. Rigorous resource management techniques such as ones presented in this thesis may help cloud providers to offer more predictable virtual machine performance and open the door for new applications that would have been difficult to run in the cloud. We would like to explore this possibility in the future.

REFERENCES

[1] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2007, pp. 146–160.

[2] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2004, pp. 111–122.

[3] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on mulcore processors," in *European Conf. on Computer systems (EuroSys)*. ACM, 2010.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 129–142.

[5] R. Pellizzoni, A. Schranzhofery, J. Cheny, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 741–746.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[7] V. Yodaiken et al., "The rtlinux manifesto," in *Proc. of the 5th Linux Expo*, 1999.

[8] P. Mantegazza, E. Dozio, and S. Papacharalambous, "Rtai: Real time application interface," *Linux Journal*, vol. 2000, no. 72es, p. 10, 2000.

[9] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*. IEEE, 1998, pp. 112–119.

[10] S. Oikawa and R. Rajkumar, "Portable rk: A portable resource kernel for guaranteed and enforced timing behavior," in *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE.* IEEE, 1999, pp. 111–120.

[11] S. Childs and D. Ingram, "The linux-srt integrated multimedia operating system: Bringing qos to the desktop," in *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE.* IEEE, 2001, pp. 135–140.

[12] *Integrity RTOS*, Green Hills Software. [Online]. Available: http://www.ghs.com/products/rtos/integrity.html

[13] *LynxOS RTOS*, lynuxworks. [Online]. Available: http://www.lynuxworks.com/rtos/

[14] *QNX Neutrino RTOS*, QNX. [Online]. Available: http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html

[15] *Wind River VxWorks RTOS*, Wind River. [Online]. Available: http://www.windriver.com/products/vxworks/

[16] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *Euromicro Conference on Real-Time Systems (ECRTS).* IEEE, 2001, pp. 199–206.

[17] T. Yang, T. Liu, E. Berger, S. Kaplan, and J. Moss, "Redline: First class support for interactivity in commodity operating systems," in *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[18] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Real-Time Systems Symposium (RTSS).* IEEE, 2006, pp. 111–126.

[19] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta, in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2011.

[20] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosaadaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[21] L. Sha, T. Abdelzaher, K. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok, "Real time scheduling theory: A historical perspective," *Real-time systems*, vol. 28, no. 2-3, pp. 101–155, 2004.

[22] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *Performance Evaluation Review*, vol. 35, no. 2, p. 42, 2007.

[23] *ARINC 653 Specification*, Aeronautical Radio Inc., 2003. [Online]. Available: http://www.arinc.com/

[24] J. Schaffer and S. Reid, "The joy of scheduling," QNX Software, Tech. Rep., 2011.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 7, no. 3, 2008.

[26] "Variable smp (4-plus-1) a multi-core cpu architecture for low power and high performance," Nvidia, Tech. Rep., 2011.

[27] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Digital Avionics Systems Conference(DASC)*. IEEE, 2007, pp. 2–A.

[28] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium(RTSS)*. IEEE, 2007, pp. 239–243.

[29] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *High-Performance Computer Architecture (HPCA)*. IEEE, 2005, pp. 340–351.

[30] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-Time Cache Management Framework for Multi-core Architectures," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013.

[31] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE, 2008, pp. 63–74.

[32] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair queuing memory systems," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2006, pp. 208–222.

[33] A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[34] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[35] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Real Time Systems Symposium*. IEEE, 1989, pp. 166–171.

[36] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[37] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Real-Time Systems Symposium (RTSS)*. IEEE, 1998, pp. 4–13.

[38] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[39] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, pp. 187–205, 2003.

[40] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making shared caches more predictable on multicore platforms," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[41] J. Nesbit, J. Laudon, and J. Smith, "Virtual private caches," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 57–68.

[42] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *European Conf. on Computer systems (EuroSys)*, 2009.

[43] X. Ding, K. Wang, and X. Zhang, "Srm-buffer: an os buffer management technique to prevent last level cache from thrashing in multicores," in *European Conf. on Computer systems (EuroSys)*. ACM, 2011, pp. 243–256.

[44] *P4080: QorIQ P4080/P4040/P4081 Communications Processors with Data Path*. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080

[45] *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R Edition*, ARM. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html

[46] *Intel®64 and IA-32 Architectures Software Developer Manuals*, Intel., 2012. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[47] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2007, pp. 251–256.

[48] M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.

[49] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed tim-criticality memory controllers," in *Design, Automation and Test in Europe (DATE)*, 2013.

[50] Z. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), in-submission*, 2013.

[51] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2008, pp. 3–14.

[52] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 5, pp. 611–624, 1998.

[53] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011, pp. 99–108.

[54] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Design Automation Conference (DAC)*, 2007.

[55] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. Lee, "A pret microarchitecture implementation with repeatable timing and competitive performance," in *Computer Design (ICCD)*. IEEE, 2012, pp. 87–93.

[56] N. Rafique, W. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2007, pp. 245–258.

[57] R. Jain, C. J. Hughes, and S. Adve, "Soft real-time scheduling on simultaneous multithreaded processors," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2002, pp. 134–145.

[58] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1. ACM, 2002, pp. 66–76.

[59] A. Fedorova, M. Seltzer, and M. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2007, pp. 25–38.

[60] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2008, pp. 220–229.

[61] S. Zhuravlev, J. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379780

[62] J. Calandrino and J. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *Euromicro Conference on Real-Time Systems (ECRTS)*. Ieee, 2008, pp. 299–308.

[63] J. Calandrino and J. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2009, pp. 194–204.

[64] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses, "Rate-based qos techniques for cache/memory in cmp platforms," in *International Conference on Supercomputing (ICS)*, 2009.

[65] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," University of Erlangen, Germany, Tech. Rep. TR-I4-97-02, July 1997. [Online]. Available: http://i30www.ira.uka.de/

[66] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," in *Symposium on Operating Systems Principles (SOSP, Poster Session)*, 1997.

[67] E. Ebrahimi, C. Lee, O. Mutlu, and Y. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," *ACM Sigplan Notices*, vol. 45, no. 3, p. 335, 2010.

[68] P. Mejia-Alvarez, E. Levner, and D. Mossé, "Adaptive scheduling server for power-aware real-time tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, p. 306, 2004.

[69] F. Gruian, "Hard real-time scheduling for low-energy using stochastic data and DVS processors," in *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM New York, NY, USA, 2001, pp. 46–51.

[70] R. Jejurikar and R. Gupta, "Optimized slowdown in real-time task systems," *IEEE Transactions on Computers (TC)*, vol. 55, no. 12, p. 1588, 2006.

[71] H. Aydin, R. Melhem, D. Mossé, and P. Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2001, pp. 225–232.

[72] P. Pillai and K. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," in *Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 89–102.

[73] X. Zhong and C. Xu, "Energy-Aware Modeling and Scheduling of Real-Time Tasks for Dynamic Voltage Scaling," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2005, pp. 10 pp.–375.

[74] E. Bini, G. Buttazzo, and G. Lipari, "Speed modulation in energy-aware real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[75] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing CPU energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst. (TECS)*, vol. 8, no. 4, pp. 1–23, 2009.

[76] J. Zhuo and C. Chakrabarti, "System-level energy-efficient dynamic task scheduling," in *Design Automation Conference (DAC)*. New York, NY, USA: ACM, 2005, pp. 628–631.

[77] H. Aydin, V. Devadas, and D. Zhu, "System-Level Energy Management for Periodic Real-Time Tasks," in *Real-Time Systems Symposium (RTSS)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 313–322.

[78] X. Fan, C. Ellis, and A. Lebeck, "The synergy between power-aware memory systems and processor voltage scaling," *Power-Aware Computer Systems*, pp. 151–166, 2005.

[79] Y. Cho and N. Chang, "Energy-Aware Clock-Frequency Assignment in Microprocessors and Memory Devices for Dynamic Voltage Scaling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1030–1040, 2006.

[80] D. Snowdon, S. Petters, and G. Heiser, "Accurate on-line prediction of processor and memoryenergy usage under voltage scaling," in *International Conference on Embedded Software (EMSOFT)*. ACM, 2007, p. 93.

[81] D. Snowdon, E. Le Sueur, S. Petters, and G. Heiser, "Koala: A platform for OS-level power management," in *European Conf. on Computer systems (EuroSys)*. ACM New York, NY, USA, 2009, pp. 289–302.

[82] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 299–316, 2002.

[83] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling and power management for portable systems," in *Design Automation Conference (DAC)*. ACM, 2001, p. 529.

[84] H. Liu, Z. Shao, M. Wang, and P. Chen, "Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 92–101.

[85] H. Cheng and S. Goddard, "Integrated device scheduling and processor voltage scaling for system-wide energy conservation," in *Workshop on Power Aware Real-time Computing (PARC)*, vol. 2, no. 7, 2005.

[86] P. Rong and M. Pedram, "Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time system," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2006, pp. 6–pp.

[87] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society Washington, DC, USA, 2008, pp. 34–44.

[88] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Multimedia Computing and Networking (MNCN)*, January 1998.

[89] G. Lipari and S. Baruah., "Greedy reclaimation of unused bandwidth in constant bandwidth servers." in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2000.

[90] M. Caccamo, G. Buttazzo, and D. Thomas, "Efficient reclaiming in reservation-based real-time systems," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2005, pp. 198–213.

[91] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2000, pp. 295–304.

[92] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "Iris: A new reclaiming algorithm for server-based real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[93] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource sharing in gpu-accelerated windowing systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011.

[94] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX Annual Technical Conference (ATC)*. USENIX, 2011.

[95] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari, "A robust mechanism for adaptive scheduling of multimedia applications," *ACM Trans. Embed. Comput. Syst. (TECS)*, vol. 10, no. 4, pp. 46:1–46:24, Nov. 2011.

[96] J. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[97] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2002, pp. 71–80.

[98] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2003, pp. 2–13.

[99] A. Jaleel, *Memory Characterization of Workloads Using Instrumentation-Driven Simulation*, 2010. [Online]. Available: http://www.glue.umd.edu/ajaleel/workload

[100] *Intel ®64 and IA-32 Architectures Optimization Reference Manual*, Intel., April 2012. [Online]. Available: http://download.intel.com/products/processor/manual/325383.pdf

[101] A. Kurdila, M. Nechyba, R. Prazenica, W. Dahmen, P. Binev, R. DeVore, and R. Sharpley, "Vision-based control of micro-air-vehicles: Progress and problems in estimation," in *Decision and Control, 43rd IEEE Conf. on*, vol. 2. IEEE, 2004, pp. 1635–1642.

[102] A. Schranzhofer, R. Pellizzoni, J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011, pp. 213–222.

[103] R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2007, pp. 73–82.

[104] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011, pp. 269–279.

[105] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[106] P. Jackson and C. Lameter, *CGROUPS*. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[107] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Circuits and Systems*, vol. 4. IEEE, 2000, pp. 101–104.

[108] S. Chakraborty, S. Künzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Design, Automation and Test in Europe (DATE)*, 2003, pp. 190–195.

[109] H. Leontyev, S. Chakraborty, and J. Anderson, "Multiprocessor extensions to real-time calculus," in *Real-Time Systems Symposium (RTSS)*. IEEE, 2009, pp. 410–421.

[110] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium (RTSS)*, 2007, pp. 49–60.

[111] A. Schranzhofer, J. Chen, and L. Thiele, "Timing analysis for tdma arbitration in resource sharing systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2010, pp. 215–224.

[112] D. Dasari, B. Andersson, V. Nelis, S. Petters, A. Easwaran, and J. Lee, "Response time analysis of cots-based multicores considering the contention on the shared memory bus," in *Trust, Security and Privacy in Computing and Communications (TrustCom)*, nov. 2011, pp. 1068 –1075.

[113] P. Turner, B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Ottawa Linux Symposium (OWLS)*, 2010.

[114] M. Stanovich, T. Baker, and A. Wang, "Experience with sporadic server scheduling in linux: Theory vs. practice," in *Real-Time Linux Workshop*, 2011.

[115] H. Shim, Y. Cho, and N. Chang, "Power saving in hand-held multimedia systems using MPEG-21 digital item adaptation," in *Workshop on Embedded Systems for Real-Time Multimedia*, 2004, pp. 13–18.

[116] *STMP3700 System-on-Chip Fact Sheet*, Freescale, October 2008. [Online]. Available: http://www.freescale.com/files/32bit/doc/fact_sheet/STMP3700FS.pdf

[117] *S3C2440A User's Manual*, Samsung, 2010. [Online]. Available: http://www.datasheetarchive.com/pdf-datasheets/Datasheets-29/DSA-568079.pdf

[118] D. M. e. Brooks, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro*, vol. 20, no. 6, pp. 26–44, 2000.

[119] *Cortex-A8 Architecture*, TI, 2010. [Online]. Available: http://wiki.davincidsp.com/index.php/Startup_shutdown_and_power_management

[120] G. Dhiman, K. Pusukuri, and T. Rosing, "Analysis of Dynamic Voltage Scaling for System Level Energy Management," *Workshop on Power Aware Computing and Systems (HotPower)*, 2008.

[121] *STMP3650 Product Overview*, Freescale, 2006. [Online]. Available: http://www.datasheet4u.com/html/S/T/M/STMP3600_Sigmatel.pdf.html

[122] *Low Voltage Intel Xeon Processor with 800 MHz System Bus*, Intel, October 2004. [Online]. Available: http://download.intel.com/design/intarch/datashts/30409701.pdf

[123] S. Eranian, *perfmon2 project website*, 2010. [Online]. Available: http://perfmon2.sourceforge.net/

[124] *K4M281633H Mobile SDRAM Datasheet*, Samsung, 2010. [Online]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/146537/SAMSUNG/K4M281633H.html

[125] F. David, J. Carlyle, and R. Campbell, "Context switch overheads on mobile device platforms," in *Workshop on Experimental Computer Science*. USENIX Association, 2007, p. 2.

[126] *The ARM Instruction Set*, ARM, 2010. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf

[127] M. Wan, Y. Ichikawa, D. Lidsky, and J. Rabaey, "An energy conscious methodology for early design exploration of heterogeneous DSPs," in *Custom Integrated Circuits Conference, 1998. Proceedings of the IEEE 1998*. IEEE, 2002, pp. 111–117.

[128] T. Šimunić, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Design Automation Conference (DAC)*. New York, NY, USA: ACM, 1999, pp. 867–872.

[129] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*. USENIX Association, 2010, pp. 1–8.

[130] J. Schad, J. Dittrich, and J. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.